OLTRE 90 ESEMPI COMMENTATI PER IMPARARE RAPIDAMENTE AD USARE IL LINGUAGGIO DEL WEB 2.0

# JAVASCRIPT BY EXAMPLE

Francesco Smelzo

# JAVASCRIPT BY EXAMPLE

**Francesco Smelzo**

*A Luigi e Dina*

# INDICE

## LINGUAGGIO

1.1 Dubbi sintattici . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .7
1.2 Lavorare com gli oggetti . . . . . . . . . . . . . . . . . . . . . . . . . . .19
1.2.1 Le funzioni . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .19
1.3 Gli array . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .26
1.4 Gli oggetti . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .28
1.5 Estensioni utili
agli oggetti predefiniti . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .33
1.6 Programmazione a oggetti . . . . . . . . . . . . . . . . . . . . . . . . .42

## OTTIMIZZAZIONE ED EFFICIENZA IN JAVASCRIPT

2.1 Ottimizzazione del codice . . . . . . . . . . . . . . . . . . . . . . . . .51
2.2 Efficienza reale degli script
in ambiente Internet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .53

## NUOVA GESTIONE DEL DOM HTML E STILI

3.1 Gestione degli eventi . . . . . . . . . . . . . . . . . . . . . . . . . . . .57
3.2 Firefox il metodo attachevent
proprio di IE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .73
3.3 Posizione
e dimensioni degli elementi . . . . . . . . . . . . . . . . . . . . . . . . .79
3.4 Table Object Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .86
3.5 Eventi . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .88
3.6 Localizzazione . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .100
3.7 Sorting di tabelle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .111

# PREMESSA

Con il Web 2.0 il paradigma dei servizi HTTP come richiesta/risposta sta a poco a poco sgretolandosi. Il client (ovvero il browser dell'utente), sempre più potente e la disponibilità di banda in aumento fanno sì che la pagina web sia un vero e proprio "miniprogramma" con una logica client/server; è quell'insieme di tecnologie conosciuto come AJAX.

Abbiamo già parlato di AJAX, in senso generale, in un altro nostro libro, in questo invece vogliamo parlare di uno dei protagonisti di AJAX: Javascript.

Javascript è un linguaggio che sta conoscendo una seconda giovinezza: nato alla fine degli anni '90 per consentire qualche piccolo effetto nelle pagine Web, adesso è diventato la colonna portante di applicazioni anche importanti.

Questo libro non vuol essere un manuale di Javascript, ma piuttosto una raccolta di consigli, risposte a dubbi frequenti, pezzi di codice utili in mille occasioni. Ovviamente si dà per scontato che il lettore abbia già un qualche background del linguaggio: quando parleremo di linguaggio non lo faremo in senso organico, ma solo per affrontare quelli che sono gli errori comuni soprattutto da parte di chi arriva a Javascript da un linguaggio con sintassi non C-Like (come ad esempio il Basic o il Pascal).

**Avvertenza: Alcuni degli script, quelli più complessi, sono riportati nel sito web** *http://www.smelzo.it/100script*

# LINGUAGGIO

## 1.1 DUBBI SINTATTICI

Javascript è il linguaggio che ormai domina incontrastato il campo dello sviluppo Web lato client, sebbene la sua sintassi non sia difficile, può a volte generare dubbi in chi si avvicina a Javascript da linguaggi sintatticamente abbastanza diversi come il Basic o il Pascal.

### 1 Il punto e virgola ci vuole o no?

I linguaggi C-Like (come C++, Java, C#,PHP ecc..) impongono che ogni istruzione sia separata da ";" come:

```
a=2;
b=3;
```

In Javascript però l'uso del punto e virgola non è obbligatorio, il codice precedente potrebbe essere scritto anche come:

```
a=2
b=3
```

In un caso però l'uso del punto e virgola diventa obbligatorio; quanto si vogliono mettere più istruzioni nello stesso rigo:

```
a=2; b=3;
```

### 2 Come si scrive il testo

Il testo inteso come valore (Stringa) in javascript si scrive tra virgolette semplici ' o doppie ":

```
var s="testo";
var s='testo';
```

più stringhe si concatenano usando l'operatore +:

```
var s = "questo " + "è " + "un " +  "testo";
```

non è possibile però usare della stessa dichiarazione sia le virgolette semplici che doppie, questo è errato:

```
var s = 'questo ' + "è " + 'un ' +  "testo";
```

Se nel testo occorre usare il tipo di virgoletta che abbiamo scelto come delimitatore bisogna anteporre il carattere \ (carattere di escape):

```
var s="\"testo\"";
var s='\'testo\'';
```

Gli accapo si scrivono usando i caratteri di escape \r\n o anche solo \n:

```
var s ="testo \n accapo";
```

## 3 Maiuscole e minuscole

Per chi viene da Visual Basic è importante ricordare sempre che Javascript è un linguaggio Case-Sensitive ovvero fa distinzione tra maiuscole e minuscole, questo passaggio ad esempio è errato:

```
var A=1;
var b= a+1;
```

## 4 Variabili globali o di funzione

Javascript consente molta libertà nella dichiarazione di variabili, esse possono essere dichiarate anche senza usare l'operatore var:

```
a=1;
è uguale a:
```

```
var a=1;
```

Spesso, all'interno di funzioni si dichiarano variabili senza l'operatore var :

```
function sayHallo (){
    s = "Hallo";
    alert(s);
}
```

facendo così la variabile s viene creata a livello globale, come se avessimo scritto:

```
    window.s = "Hallo";
```

in questo modo la variabile viene distrutta solo al termine dell'esecuzione del codice, non quando finisce la funzione. Oltretutto questo modo di programmare porta a commettere degli errori di non facile individuazione, si veda questo codice:

```
function prima(){
    for(i==0;i<10;i++){
        seconda();
}
}
function seconda(){
    for(i==0;i<5;i++){
        alert(i);
    }
}
```

In questo caso la variabile i è dichiarata a livello globale, ma viene usata sia nel ciclo for della funzione prima che in quello della funzio-

ne seconda, in questo modo quando viene eseguita seconda per la prima volta i vale 4, quindi il secondo passaggio del ciclo for di prima non parte da 1 come ci si aspetterebbe, ma da 4 e così via. Di fatto anzi la chiamata di seconda porterebbe a un ciclo infinito in quanto poiché i esce da seconda a 4 nel ciclo di prima non si verifica mai la condizione i<10. Quindi usare sempre l'operatore var nella dichiarazione:

```
function prima(){
	for(var i==0;i<10;i++){
		seconda();
}
}
function seconda(){
	for(var i==0;i<5;i++){
		alert(i);
	}
}
```

## 5 Assegnazione e comparazione

Il programmatore di Visual Basic che si cimenta con Javascript cade spesso nell'equivoco tra assegnazione e comparazione, in questo linguaggio infatti non c'è differenza tra i due operatori (sempre "="), quindi posso scrivere:

```
Dim a,b
a=1
b=2
If a=b Then
	b=3
End If
```

Il valore di b qui non sarà mai uguale a 3 perché il valore di b è diverso da quello di a. Traducendo il codice in Javascript accade di scrivere:

```
var a,b;
a=1

b=2
if(a=b){
      b=3;
}
```

Eseguendo il passaggio ci accorgeremo che in Javascript b è uguale a 3! Com'è possibile? L'errore, piuttosto comune è dato dal fatto che la comparazione in Javascript va fatta con l'operatore "==" mentre "=" è l'operatore di assegnazione.
In pratica dire (a=b) significa assegnare ad a il valore di b (operazione che dà comunque true, per cui if viene eseguita).
Quindi occorreva scrivere:

```
var a,b;
a=1
b=2

if(a==b){
      b=3;
}
```

## Conoscere il tipo di oggetto

In Javascript tutto viene visto come 'Oggetto', tuttavia abbiamo alcuni oggetti 'specializzati' come Array, Date, Number, Boolean, String e Object stesso (per rappresentare tutti gli altri oggetti). È possibile determinare il tipo di oggetto contenuto in una variabile con l'operatore typeof che restituisce le stringhe : "number", "string", "boolean", "object", "function", e "undefined"

## 6 La variabile è una funzione?

```
function isFunction(f){
    return  (typeof(f)=='function');
}
//Esempio:
var s ="stringa";

function myFunc (){
    return "myFunc";
}
var aFunction = function  (){ //funzione assegnata a variabile
    return null;
}
document.write (isFunction(myFunc)); // true
document.write (isFunction(aFunction)); // true
document.write (isFunction(s)); // false
```

In javascript anche le funzioni sono oggetti come gli altri per cui è possibile assegnarle alle variabili

## 7 La variabile è una stringa?

```
function isString(obj){
    return (typeof(obj)=='string');
}
//Esempio:
var s ="stringa";

function myFunc (){
    return "myFunc";
}

document.write (isString(s)); // true
document.write (isString(myFunc)); // false
```

## 8 La variabile è un valore booleano?

```
function isBool(obj){
    return (typeof(obj)=='boolean');
    }
//Esempio:
var b = true;
var bs = "true";
var bNumber = 1;
document.write (isBool(b)); // true
document.write ('<br/>');
document.write (isBool(bs)); // false
document.write (isBool(bNumber)); // false
```

Notare come venga riconosciuta come Boolean solo la variabile con assegnato un valore di questo tipo, non una stringa di valore “true” o “false”, né un numero assimilabile (0 o 1). Per coprire anche queste casistiche dovremo ampliare la funzione:

```
function isRealBool(obj){
    if(isString(obj)) {
        var v = obj.toLowerCase();
        return (v=="true" || v=="false" || v=="0" || v=="1");
    }
    if(typeof(obj)=='number') {
        return (obj==0 || obj==1);
    }
    return (typeof(obj)=='boolean');
}
```

## 9 La variabile è un numero?

```
function isNumber(obj){
    return (typeof(obj)=='number');
}
```

```
//Esempio:
document.write (isNumber("1")); // false
document.write (isNumber(1)); // true
```

Anche qui vediamo che il valore non viene riconosciuto come numero quando è rappresentato da una stringa, per coprire anche questo caso possiamo scrivere :

```
function isRealNumber(obj){
	var n = new Number(obj);
	return (!isNaN(n));
}
//Esempio:
document.write (isRealNumber(1)); // true
document.write (isRealNumber("s")); // false
document.write (isRealNumber("2.0")); // true
10.	La variabile è un oggetto?	1/5
function isObject (obj){
	return (typeof(obj)=='object');
}
//Esempio:
var obj = new Object;
var s= "";
document.write (isObject(obj)); // true
document.write (isObject(s)); // false
```

Purtoppo però nel concetto di "Oggetto" typeof comprende anche due importanti tipi come gli Array (matrici) e le Date, quindi per ispezionare l'oggetto per determinare se è un Array o una data occorre far ricorso ad altre funzioni, di seguito illustrate.

## 11 La variabile è un Array?

Qui occorre far ricorso all'operatore instanceof che confronta

l'oggetto con un tipo e restituisce il valore true se l'oggetto è di quel tipo. Quindi:

```
function isArray(obj){
    return obj instanceof Array;
}
//Esempio:
var obj = new Object;
var arr = new Array;
document.write (isArray(obj)); // false
document.write (isArray(arr)); // true
```

## 12 La variabile è una data?

Analogamente, se vogliamo scoprire se una variabile è di tipo Date possiamo utilizzare:

```
function isDate(obj){
    return obj instanceof Date;
}
//Esempio:
var date = new Date(2007,5,1);
var sdate = "01/06/2007";
document.write (isDate(date)); // true
document.write (isDate(sdate)); // false
```

## 13 La variabile è veramente una data?

la funzione che abbiamo visto al punto 12 riconosce soltanto le istanze del tipo Date e non, ad esempio, le stringhe che contengono una data. Realizzare una funzione che riconosce la data anche nelle stringhe è un po' più complesso. Il parsing di una data richiede l'uso delle regular expressions, la difficoltà sta soprattutto nei vari formati che possono essere usati per rappresentare le date, qui prendiamo in considerazione i più diffusi

(giorno-mese-anno, mese-giorno-anno e anno-mese-giorno) espressi in forma numerica. Il parametro format della funzione dovrà essere appunto 'gma', 'mga' o 'amg' quello di default è 'gma').

```
function isRealDate(obj,format){
    //format puo' essere : gma,mga o amg
    //i separatori ammessi sono : \ / . -
    if(isDate(obj))return true; //obj è di tipo Date
    var s = obj.toString();
    var patterns = new Array();
//pattern per GMA e MGA
    patterns.push(/^(\d{1,2})(\/|-|\.|\\)(\d{1,2})(\/|-|\.|\\)(\d{4})$/);
//pattern per AMG
    patterns.push(/^(\d{4})(\/|-|\.|\\)(\d{1,2})(\/|-|\.|\\)(\d{1,2})$/);
    var day,month,year;
    switch (format) {
        case 'gma':
            var matchArray = s.match(patterns[0]);
            if(matchArray==null) return false;
            day = matchArray[1];
            month = matchArray[3];
            year = matchArray[5];
            break;
        case 'mga':
            var matchArray = s.match(patterns[0]);
            if(matchArray==null) return false;
            day = matchArray[3];
            month = matchArray[1];
            year = matchArray[5];
            break;
        case 'amg':
            var matchArray = s.match(patterns[1]);
            if(matchArray==null) return false;
```

```
                day = matchArray[5];
                month = matchArray[3];
                year = matchArray[1];
                break;
            default:
                return isRealDate(obj,'gma');
        }
        if (day < 1 || day > 31) return false; //il giorno deve essere da 1 a 31
        if (month < 1 || month > 12) return false; //il mese deve essere
                                                                    da 1 a 12
        //questi mesi hanno 30 giorni:
        if ((month==4 || month==6 || month==9 || month==11) && day==31)
                                                                return false;
        if (month == 2) { //febbraio
            //verifica se l'anno è bisestile
            var bisestile = (year % 4 == 0 && (year % 100 != 0 || year % 400
                                                                        == 0));
            if (day > 29 || (day==29 && !bisestile)) return false;
        }
        return true; //la data è valida
}
//Esempio:
var date = new Date(2007,5,1);
var sdate = "01/06/2007";
var nodate = "non è una data";
document.write (isRealDate(date,'gma')); // true
document.write ('<br/>');
document.write (isRealDate(sdate,'gma')); // true
document.write ('<br/>');
document.write (isRealDate(nodate,'gma')); // false
```

## 14 La variabile è definita?

A volte può essere necessario sapere se una determinata varia-

bile è stata o meno dichiarata, l'operatore typeof in tal caso può essere usato per verificare l'oggetto:

```
var exists = (typeof(variabileInesistente)!="undefined")
document.write (exists); // false
```

A prima vista si può essere tentati di generalizzare l'operazione in una funzione come:

```
function isDefined(obj){
      return  (typeof(obj)!='undefined');
}
```

Tuttavia questa funzione in questo caso genererebbe errore : isDefined(variabileInesistente); //Errore. Ciò succede perché non possiamo passare come argomento di una funzione un riferimento non definito, tuttavia, per non dovere usare sempre l'operatore typeof, possiamo ricorrere ad un piccolo trucco; le variabili a livello globale, non sono altro che proprietà dell'oggetto primario window, quando si dichiara una variabile all'esterno di una funzione:

```
var myValue = “ciao”;
è, in pratica, equivalente a:
window.myValue = “ciao”;
```

E poiché, come vedremo meglio anche più avanti, le proprietà possono essere espresse anche sotto forma di Array associativi:

```
window.myValue = “ciao”;
è equivalente a:
window['myValue'] = “ciao”;
```

Sfruttando questa caratteristica è possibile scrivere la funzione:

```
function isDefined (name){
    return (window[name]!=null);
    //Oppure return (name in window);
}
//Esempio:
var s = "";
document.write(isDefined('variabileInesistente'));// false
document.write(isDefined('s'));// true
```

da usare quindi passando come argomento il nome del riferimento da testare (sotto forma di stringa).



# 1.2 LAVORARE CON GLI OGGETTI

## 1.2.1 Le funzioni

### 15 Dichiarare una funzione

Le funzioni in javascript sono un oggetto veramente multiforme, ciò si può vedere anche dai modi in cui si possono dichiarare:

```
// come oggetto implicitamente creato (metodo più comune)
function myFunc(name){
    return "Hello " + name;
}
// equivalente : assegnazione di una var
var myFunc = function  (name){
    return "Hello " + name;
}
// equivalente : assegnazione al contesto globale
window.myFunc = function  (name){
    return "Hello " + name;
}
```

```
// equivalente : assegnazione al contesto globale
// come elemento dell'array di proprietà globale
window['myFunc'] = function  (name){
      return "Hello " + name;
}
// dichiarazione con il costruttore Function
// prima viene indicato il nome degli argomenti e poi il corpo
var myFunc = new Function('name', 'return "Hello " + name;');
```

Dagli esempi è facile comprendere come la funzione sia, per javascript, un oggetto proprio come qualsiasi altro (stringa, data ecc…), l'unica particolarità sintattica è nel modo di dichiarazione. Da comprendere bene la differenza tra riferimento ad una funzione e chiamata della funzione stessa :

```
//riferimento alla funzione
var myFuncRef = myFunc;
//chiamata della funzione
var value = myFunc('Mario');
//uguale anche a
var value = myFuncRef('Mario');
```

cioè l'utilizzo delle parentesi ( ) indica che la funzione viene eseguita e quindi rappresenta il valore di ritorno espresso dalla parola chiave return, se la funzione non restituisce nessun valore il risultato sarà null. La funzione, come vedremo più avanti, può avere anche un valore di costruttore o meglio modello per la costruzione di oggetti.

## 16 Funzioni nidificate

Una volta compreso che la funzione è un oggetto come gli altri non ci stupirà scoprire che una funzione possa essere dichiarata ed usata all'interno di un'altra:

```
function mainFunction (name){
	function aggiungiParola (parola){
		return name + ' ' + parola;
	}
	var parole = new Array();
	parole[0] = 'programmatore';
	parole[1] = 'consulente';
	parole[2] = 'professionista';
	var result = '';
	for(var i=0;i<parole.length;i++){
		result += aggiungiParola(parole[i] + '<br/>');
	}
	return result;
}
document.write(mainFunction('Mario'));
/*
```

scrive :

```
Mario programmatore<br/>Mario consulente<br/>Mario professionista
*/
```



È interessante notare come l'argomento della funzione principale mainFunction, ovvero name, mantenga il suo significato anche all'interno della funzione nidificata aggiungiParola. L'utilità delle funzioni nidificate è ben superiore a quanto può apparire : si pensi infatti che javascript non dispone di indicatori di visibilità come altri linguaggi (public, private, protected ecc…) quindi nidificare una funzione dentro l'altra è l'unico modo per mantenerla private, impedendo quindi che possa essere chiamata fuori dal contesto in cui è dichiarata.

## 17 Funzioni di callback

Il fatto che una funzione sia un oggetto porta con sé anche

un'altra conseguenza: una funzione può essere passata come riferimento, ad esempio come argomento di un'altra funzione.

```
//funzione principale : aggiunge giorni alla data
// e demanda ai possibili formatCallback la formattazione della risposta
function addDays (date,days,formatCallback){
    var addMilliseconds = function(dt,ms){
      return new Date(new Date().setTime(dt.getTime() + (ms)));
    }
    var DAYS_UNIT = 86400000;      // 24 * 60 * 60 * 1000
    var newDate = addMilliseconds(date, days * DAYS_UNIT);
    var d,m,y;
    d = newDate.getDate();
    m = (newDate.getMonth() + 1);
    y = newDate.getFullYear();
    return formatCallback(d,m,y);
}
//primo callback
function italianFormat(d,m,y){
    return d + "/" + m + "/" + y;
}
//secondo callback
function englishFormat (d,m,y){
    return m + "/" + d + "/" + y;
}
//Esempio:
var dt = new Date();
document.write(addDays(dt,10,italianFormat)); //data in italiano
document.write(addDays(dt,10,englishFormat)); //data in inglese
```

In questo caso, la funzione principale (addDays) svolge tutto il lavoro di calcolo restando al tempo stesso flessibile e riutilizzabile poiché per la formattazione dei risultati si affida alla funzione passata

come argomento di callback. Tipicamente l'utilizzo di questo costrutto si ha nelle librerie javascript, quando non si possono prevedere a priori tutti gli utilizzi del risultato di una funzione e si preferisce quindi demandare all'implementatore la gestione del risultato.

## 18 Funzioni ad argomenti liberi

In javascript gli argomenti (o parametri) sono tutti opzionali, nel senso che se alla funzione vengono associati un certo numero di argomenti e in fase di chiamata ne vengono passati solo alcuni non viene generato a priori nessun errore. Ad esempio, se abbiamo una funzione :

```
function msg (a,b){
    alert(a);
}
```

possiamo tranquillamente richiamare la funzione usando un solo parametro:

```
msg('Hello');
```

oppure usandone più di quelli originariamente previsti:

```
msg('Hello','Mario',10);
```

L'oggetto Function dispone di una proprietà chiamata arguments che consente di recuperare, in forma di Array, tutti gli argomenti passati, per cui è possibile anche non dichiarare nessun argomento e recuperarli poi nel corpo della funzione:

```
function sumAll (){
    var n= 0;
    for(var i=0;i<arguments.length;i++){
        var current = new Number(arguments[i]);
```

```
            if(!isNaN(current)) n += current;
      }
      return n;
}
//Esempio:
document.write(sumAll(3,6,10));// = 19
document.write(sumAll(3,6));// = 9
document.write(sumAll(3));// = 3
document.write(sumAll());// = 0
```

## 19 Testare l'esistenza di argomenti in una funzione

Se è vero che la possibilità di usare un numero non limitato a priori di argomenti aumenta la flessibilità è anche vero che usare argomenti che potrebbero non esistere ci potrebbe esporre all'inconsistenza del codice, per cui prima di usare un argomento indefinito, sarebbe preferibile testarne l'esistenza e magari fornire un valore di default, un metodo può essere:

```
function msg (){
      var part1 = arguments[0] || "null";
      var part2 = arguments[1] || "null";
      alert(part1 + part2);
}
```

dove si riassegnano gli argomenti previsti a variabili interne con un'operazione booleana  di test che fornisce contemporaneamente un valore di default. Questa tecnica è molto utilizzata anche se ha un limite : arguments[n] sarà scartato sia nel caso non sia definito sia in quello che assuma il valore booleano false. Quindi, nel nostro esempio la chiamata:

```
msg('ciao ', false);
```

non darà il messaggio "ciao false" come ci si potrebbe aspettare, ma "ciao null"! Se si prevedono quindi argomenti che possono avere anche valori booleani conviene adottare una strategia più "difensiva", come :

```
function msg (){
    var args = arguments;
    function getArg(index,defaultValue){
        if(args[index]==null) return defaultValue;
        return args[index];
    }
    var part1 = getArg(0,"null");
    var part2 = getArg(1,"null");
    alert(part1 + part2);
}
```

questo ci garantisce meglio, ma ci costringerebbe comunque a definire una funzione interna di recupero degli argomenti per ogni funzione dichiarata, la soluzione sta nell'intervenire direttamente sul prototype dell'oggetto Function per implementare un nuovo metodo (vedremo meglio più avanti cos'è il prototype):

```
Function.prototype.getArg = function (index,defaultValue){
    var args = this.arguments;
    if(args[index]==null) return defaultValue;
    return args[index];
}
```

In questo modo in ogni funzione che scriviamo potremo utilizzare getArg (preceduto dal nome della funzione stessa):

```
function msg (){
    var part1 = msg.getArg(0,"null");
```

```
    var part2 = msg.getArg(1,"null");
    alert(part1 + part2);
}
```

# 1.3 GLI ARRAY

## 20 Creare un Array

In Javascript gli Array (o matrici) sono un oggetto importantissimo visto che anche gli altri oggetti non sono altro che membri di Array associativi (un Array associativo è quello composto da coppie nome"valore). Infatti nel contesto di una pagina web dichiarare:

```
var nome='mario';
```

o

```
window['nome']='mario';
```

è esattamente la stessa cosa visto che le variabili, dichiarate al di fuori di funzioni, sono comunque parte dell'Array globale.
Più comunemente comunque gli Array sono nella forma indicizzata (da 0 a n). La creazione e l'aggiunta di elementi all'Array è semplicissima:

```
var a = new Array;
a[0] = "Rosso" ;
a[1] = "Giallo" ;
a[2] = "Verde" ;
```

Ma esiste anche una forma contratta, comunissima, con:

```
var a = [];
```

Questa forma di dichiarazione consente anche di aggiungere direttamente gli elementi:

```
var a = ["Rosso", "Giallo", "Verde"];
```

## 21 Manipolare gli Array

La gestione degli Array è assistita da alcuni metodi base.
Con push è possibile accodare un elemento ad un Array:

```
a.push('Azzurro'); //aggiunge un elemento
```

Con l'operatore delete invece  è possibile cancellare un elemento:

```
delete a[3];
```

Comunque attenzione perché delete cancella l'elemento ma non ridimensiona l'Array, per cui se cancello l'elemento con indice 3 di un Array con quattro elementi (da 0 a 3 appunto), la lunghezza dell'Array sarà sempre 4 e l'ultimo valore rimarrà undefined. La rimozione di un elemento e il ridimensionamento dello stesso si ha invece con splice :

```
a.splice(3,1);
```

dove il primo parametro è l'indice dove si trova l'elemento da cancellare ed il secondo rappresenta il numero di elementi da cancellare partendo da questo indice. L'aggiunta degli elementi di un Array ad un altro si ha invece con concat:

```
var arr1 = ["giallo","rosso"];
var arr2 = ["verde","azzurro"];
var arr3 = arr1.concat(arr2);
// arr3 = ["giallo","rosso", "verde","azzurro"];
```

Se l'Array è composto da stringhe può risultare particolarmente utile il metodo join che restituisce una stringa formata dalle stringhe che compongono l'Array unite dal separatore definito come parametro oppure da virgole:

```
var a = ["Rosso", "Giallo", "Verde"];
var s = a.join(); // risultato = Rosso,Giallo,Verde
var s = a.join(';'); // risultato = Rosso;Giallo;Verde
```

Altre funzioni utili sono : **pop** (restituisce l'ultimo elemento dell'Array rimuovendolo), **shift** (restituisce il primo elemento dell'Array rimuovendolo), **unshift** (aggiunge uno o più elementi all'inizio dell'Array), **slice** (restituisce un gruppo di elementi di un Array), **reverse** (rovescia l'ordine degli elementi di un Array), **sort** (ordina gli elementi di un Array). Da non dimenticare poi la proprietà length che restituisce la lunghezza dell'Array, attenzione però perché length restituisce la lunghezza calcolata a partire da 1, poiché gli Array in Javascript sono a base 0 se ho un Array con indice da 0 a 3 length sarà 4.

# 1.4 GLI OGGETTI

## 22 Creare un oggetto

In Javascript la creazione e l'estensione di un oggetto è semplicissima, basta dichiararlo e aggiungervi proprietà e metodi desiderati. Il modo più semplice di creare un oggetto è:

```
//creare un oggetto con new
var obj = new Object;
```

esiste anche una forma contratta, resa famosa dalla notazione JSON:

```
//notazione contratta
```

```
var obj = {};
```

All'oggetto creato, come vedremo, è possibile associare proprietà e metodi a piacere, tuttavia è anche possibile definire una funzione "costruttore" per stabilire in anticipo quali saranno i membri dell'oggetto o magari per consentire di passare degli argomenti in fase di costruzione con l'operatore new.

```
//costruttore con argomenti
function Auto (posti, marca, targa){
// this rappresenta l'oggetto che verrà creato
    this.posti = posti;
    this.marca = marca;
    this.targa = targa;
}

//dichiaro l'oggetto utilizzando il modello Auto
var auto1 = new Auto (4,'Fiat 500','TO23443');
var auto2 = new Auto (4,'Citroen 2 cv','DX234ED');

//utilizzo una proprietà
document.write(auto1.marca);
```

Da notare che, a differenza di quanto si potrebbe credere, Auto non è un Tipo a se stante, ma semplicemente un modello per creare un oggetto con determinate caratteristiche, scrivere infatti :

```
var auto1 = new Auto (4,'Fiat 500','TO23443');
o scrivere
var auto1 = new Object;
auto1.posti = 4;
auto1.marca = 'Fiat 500';
auto1.targa ='TO23443';
```

è perfettamente la stessa cosa.

## 23 Aggiungere proprietà e metodi ad un oggetto

Ad un oggetto è sempre possibile aggiungere metodi e proprietà direttamente, anche quando non previsti dal modello.

```
var auto1 = new Auto (4,'Fiat 500','TO23443');
auto1.prezzo = 3000; // aggiungo proprietà
auto1.prezzoAlto = function (){ //aggiungo un metodo
    return (this.prezzo > 10000);
}
document.write(auto1.prezzoAlto()); //false
```

esiste anche una forma contratta per dichiarare proprietà e metodi direttamente in fase di creazione:

```
var auto1 = {
posti : 4,
marca : 'Fiat 500',
targa : 'TO23443',
prezzo : 3000,
prezzoAlto : function (){ return (this.prezzo     > 10000)}
};
```

in questo caso l'operatore di assegnazione non è più "=", ma ":" ed il separatore (obbligatorio) non è ";" ma ",". I nomi delle proprietà e metodi possono anche essere espressi (nel modo adottato dalla notazione JSON) in forma di stringa:

```
var auto1 = {
'posti' : 4,
'marca' : 'Fiat 500',
```

```
'targa' : 'TO23443',
'prezzo' : 3000,
'prezzoAlto' : function (){ return (this.prezzo     > 10000)}

};
```

In questo modo però definiamo proprietà e metodi per un oggetto specifico, se quest'oggetto è stato creato in base ad un modello proprietà e metodi non apparterranno agli oggetti successivamente creati con quel modello. Perché ciò avvenga è necessario intervenire sul modello e non sulla singola istanza, questo è possibile facendo riferimento alla proprietà prototype :

```
//costruttore
function Auto (posti, marca, targa){
    this.posti = posti;
    this.marca = marca;
    this.targa = targa;
}
//utilizzo di prototype
Auto.prototype.prezzo = 0; //aggiungo una proprietà
Auto.prototype.prezzoAlto = function (){ //aggiungo un metodo
    return (this.prezzo > 10000);
}
//implementazione
var auto2 = new Auto (4,'Citroen 2 cv','DX234ED');
auto2.prezzo = 11000;
document.write(auto2.prezzoAlto()); //true
```

Da notare che prototype serve soprattutto per l'aggiunta successiva di metodi e proprietà (o per migliorare la leggibilità del codice), essi possono infatti essere inseriti direttamente nel costruttore:

```
function Auto (posti, marca, targa){
	this.posti = posti;
	this.marca = marca;
	this.targa = targa;
	this.prezzo = 0;
	this.prezzoAlto = function (){
		return (this.prezzo > 10000);

	};
}
```

## 24 Estendere gli oggetti predefiniti

Nello stesso modo in cui è possibile estendere, con prototype, un modello da noi definito, è anche possibile estendere gli oggetti di base di javascript come Array, Function, String, Date, Number e lo stesso Object, in maniera da disporre di funzionalità aggiuntive originariamente non previste. Ad esempio, vediamo come implementare il metodo trim (che cancella gli spazi bianchi all'inizio e alla fine di una stringa) .

```
String.prototype.trim=function () {
return this.replace(/\s+$|^\s+/g,"");

}
//Esempio:
var s = "        stringa            ";
document.write(s.trim()); // "stringa"
```

da notare come all'interno del metodo this rappresenta l'istanza corrente della stringa stessa. Estendendo l'oggetto con prototype, nel contesto corrente, tutte le stringhe disporranno del metodo trim(). Seguendo questo modello vediamo di seguito alcune estensioni particolarmente utili agli oggetti predefiniti.

# 1.5 ESTENSIONI UTILI AGLI OGGETTI PREDEFINITI

## 25 Trovare l'indice di un elemento

L'oggetto Array manca di una funzione che consenta di trovare l'indice di un elemento presente al suo interno, ma possiamo crearla noi :

```
Array.prototype.indexOf= function(e){
    var i=this.length;
    while(--i>-1){
        if(this[i]==e) return i; //posizione
        }
    return i; // ritorna -1 (non trovato)
}
//Esempio:
var a = new Array;
a[0] = "Ciao";
a[1] = "a";
a[2] = "tutti";
document.write(a.indexOf('a')); // = 1
document.write(a.indexOf('tutti')); // = 2
document.write(a.indexOf('altro')); // = -1 non trovata
document.write(a.indexOf('ciao')); // = -1 il confronto è case sensitive
```

## 26 Trovare l'indice di un elemento in maniera case-insensitive

Come abbiamo visto, il confronto, in questo caso è case sensitive (fa differenza tra maiuscole e minuscole), per gli Array di stringhe però può essere comodo anche di disporre di un metodo case insensitive come :

```
Array.prototype.indexOfString= function(e){
    e = e.toString().toLowerCase();
```

```
    var i=this.length;
    while(--i>-1){
        if(this[i].toString().toLowerCase()==e) return i; //posizione
        }
    return i; // ritorna -1 (non trovato)
}

//Esempio:
document.write(a.indexOfString('ciao')); // = 0 il confronto
                                                    è case insensitive
```

## 27 Ordinare un Array in maniera case-insensitive

La funzione sort dell'Array è case-sensitive, se l'applichiamo a una matrice di stringhe con caratteri misti (maiuscoli/minuscoli) potremo ottenere risultati inaspettati :

```
var a = ["Carlo","anna","roberto","Antonio"];
a.sort();
```

ci aspetteremmo l'ordinamento :

- anna
- Antonio
- Carlo
- roberto

Mentre in realtà avremmo:

- Antonio
- Carlo
- anna
- roberto

Questo succede perché l'ordinamento avviene con il confronto del valore numerico dei caratteri (il carattere “A” per esempio equivale a 65, mentre “a” è 97). A questo si può porre rimedio con una funzione di sorting personalizzata da associare al prototipo Array:

```
Array.prototype.sortInsensitive = function  (){
    this.sort (function (elm1,elm2){
        var v1 = elm1.toString().toLowerCase();
var v2 = elm2.toString().toLowerCase();
        if(v1==v2) return 0;
        else return (v1>v2) ? 1:-1;
    });
}
```

che richiama il sort standard applicandogli però una funzione di callback che verrà richiamata in fase di comparazione tra i due elementi operando un confronto case-insensitive.

## 28 Ordinare un Array con più opzioni

Sempre agendo sul prototipo di Array è possibile dotare quest'ultimo di una funzione di sorting un po' più completa che consenta di specificare la modalità (case sensitive o case insensitive) e la direzione (ascendente o discendente):

```
Array.prototype.sortFull = function  (insensitive,desc){

    var isInsensitive = insensitive || true; //default true
    var isDesc = desc || false; //default false
    if(isInsensitive) this.sortInsensitive();
    else this.sort();

    if (isDesc) this.reverse();
}
```

richiamabile semplicemente con:

```
a.sortFull(true,false); //sorting case insensitive ascendente
a.sortFull(true, true); //sorting case insensitive discendente
```

## 29 Trasformare una collection in Array

In Javascript non tutto quello che sembra un Array in realtà lo è davvero, ci sono molti oggetti (come ad esempio l'oggetto arguments di Function, che abbiamo visto al punto 19), che contengono altri oggetti, dispongono della proprietà length e possono essere scorsi per indici con cicli for, ma non dispongono dei metodi dell'oggetto Array perché non derivano da esso.
Sono le cosiddette liste, o collections, che nel DOM sono molto diffuse. Non essendo Array, se ad esempio applichiamo il metodo pop (che restituisce l'ultimo elemento di un Array) all'oggetto arguments di una funzione avremmo un errore:

```
function fx (){
      alert(arguments.pop())
      }
fx(1,2,3,4);
```

infatti pop non è membro di arguments in quanto quest'ultimo non ha il costruttore Array. Per poter ricavare un Array partendo da una collection definiamo quindi un metodo statico che potremmo associare all'oggetto Array stesso:

```
//prevediamo anche una sintassi breve per poter usare anche $A()
//oltre che Array.from()
var $A = Array.from = function(iterable) {
  if (!iterable) return []; //se l'argomento è nullo ritorna array vuoto
//se l'oggetto in argomento dispone di un metodo toArray
                                                    ritorna il risultato
```

```
  if (iterable.toArray) {
return iterable.toArray();
  }
  else {
//altrimenti aggiunge i membri della collection a un nuovo Array
//e ritorna quest'ultimo
    var results = [];
    for (var i = 0; i < iterable.length; i++)
      results.push(iterable[i]);
    return results;
  }
}
```

In questo modo, applicando la trasformazione, la funzione vista in precedenza funzionerà correttamente con:

```
function fx (){
       alert($A(arguments).pop())
       }
fx(1,2,3,4);
```

## 30 Aggiungere il metodo foreach agli Array

Uno dei costrutti più usati in Javascript è il classico ciclo for per scorrere gli Array:

```
var a = [1,2,3,4,5];
for(var i=0;i<a.length;i++){
       //codice ...
}
```

in realtà si tratta di associare l'esecuzione di codice per ogni elemento dell'Array. La semplificazione di questo pattern la possiamo ottenere aggiungendo il metodo foreach al prototipo di

Array fornendo a quest'ultimo una funzione di callback da utilizzare internamente in fase di scorrimento degli elementi:

```
Array.prototype.foreach = function(callback){
    if(typeof(callback)=="function"){
        for(var i=0; i<this.length;i++) {
            callback(this[i]);
        }
    }
}
```

in questo modo il frammento di codice che abbiamo visto all'inizio può essere riscritto, più elegantemente:

```
var a = [1,2,3,4,5];
a.foreach(function (element){
      //codice ...
});
```

La funzione naturalmente può essere anche definita esternamente è passata come riferimento:

```
function parseElement (element){
      //codice ...
}
var a = [1,2,3,4,5];

a.foreach(parseElement);
```

## 31 Aggiungere il Trim alle stringhe

In Javascript non c'è una funzione che togli gli spazi vuoti prima o dopo una stringa. Niente paura, è facile implementarla estendendo il prototipo di String:

```
String.prototype.Trim=function () {
return this.replace(/\s+$|^\s+/g,"");
}
```

Così potremmo usare:

```
var s = "     test    ";
s.Trim(); // = "test"
```

## 32 Con quale carattere inizia la stringa...

Volete sapere al volo se una stringa inizia con un dato carattere? Estendiamo String:

```
String.prototype.startsWith=function(s){
   return  (this.substring(0,s.length).toLowerCase()==s.toLowerCase());
}
```

Quindi:

```
var s="casa";
var b = s.startsWith("c");//true
```

## 33 ...e con quale finisce.

Naturalmente è possibile anche implementare l'endsWith:

```
String.prototype.endsWith=function(s){
   return  this.substr(this.length - s.length).toLowerCase ()==s.toLower
Case ();
}
Quindi:
var s="casa";

var b = s.startsWith("a");//true
```

## 34 Comparazione di stringhe in maniera case-insensitive

Stanchi di trasformare le stringhe con toLowerCase o toUpperCase in fase di comparazione tra due stringhe? Quello che ci vuole è una funzione come questa da aggiungere al prototipo String:

```
String.prototype.invariantCompare = function (strCompare){
    if (this.toLowerCase()>strCompare.toLowerCase()) return 1;
    else if (this.toLowerCase()<strCompare.toLowerCase()) return -1;
    else return 0;
}
```

la funzione dà 1 nel caso in cui la prima stringa sia alfabeticamente posteriore alla seconda, -1 in caso sia anteriore e 0 se sono uguali. Questa funzione dà un risultato di questo tipo per essere utilizzata nell'ordinamento di Array di stringhe, se ci serve un valore booleano basta definire anche:

```
String.prototype.equals = function (strCompare){
    return this.invariantCompare(strCompare)==0;
}
```

## 35 Formattare le date

Sentiamo il bisogno di una funzione che ci consenta di formattare facilmente le date? Ebbene, nulla di più facile che estendere il prototipo Date con la funzione format:

```
Date.prototype.format = function (dateFormat){
    var s = dateFormat;
    s=s.replace(/y+/,this.getFullYear());
    s=s.replace(/M+/,this.getMonth() + 1);
    s=s.replace(/d+/,this.getDay());
    s=s.replace(/h+/,this.getHours());
```

```
        s=s.replace(/m+/,this.getMinutes());
        s=s.replace(/s+/,this.getSeconds());
        return s;
}
```

Così potremmo usare:

```
var dt = new Date();
alert(dt.format("d/M/y h:m:s"));
```

## 36 Arrotondamenti

Anche l'oggetto Number si presta a interessanti estensioni, una funzione utile potrebbe essere quella degli arrotondamenti con precisione decimale:

```
Number.prototype.round = function (precision) {
        var number = this;
    if (precision == undefined) precision = 2;
    var sign = (number < 0) ? -1 : 1;
    var multiplier = Math.pow(10, precision);
    var result = Math.abs(number);
    result = Math.floor((result * multiplier) + .5000001) / multiplier;
    if (sign < 0) result = result * -1;

    return result;
}
```

Un esempio d'uso è:

```
n=1.5648562;
document.write(n.round(3));

//risultato 1,565
```

# 1.6 PROGRAMMAZIONE A OGGETTI

## 37 Implementare l'ereditarietà degli oggetti

In Javascript spesso i programmatori abituati ad altri linguaggi rimangono disorientati di fronte alla mancanza di alcuni costrutti, in particolare le classi. Javascript non si basa infatti sulle classi, ma sui prototipi, come abbiamo già visto negli scripts precedenti. In realtà l'approccio agli oggetti di Javascript ha una flessibilità senza pari, visto che consente di creare metodi e proprietà "al volo" semplicemente dichiarandoli, senza bisogno di costrutti particolari. In realtà le classi portano con sé alcuni indubbi vantaggi, tra cui, uno dei principali è l'ereditarietà, cioè la possibilità per una classe di ereditare metodi ed oggetti da un'altra classe, per cui è possibile definire degli oggetti astratti da cui altri oggetti ereditano proprietà comuni. Per focalizzare meglio il concetto pensiamo a dover scrivere un programma per la gestione di diversi tipi di veicoli, sarebbe quindi opportuno definire la classe astratta Veicolo con alcune proprietà comuni:

Veicolo
"marca
"tipologia

Poi potremmo implementare una classe Auto che eredita da Veicolo ed oltre alle sue proprietà ha anche targa:

Auto (Veicolo)
"marca (da Veicolo)
"tipologia (da Veicolo)
"targa

e così via …

Il fatto che in Javascript non ci siano le classi non significa affatto che non possiamo implementare l'ereditarietà, un modo classico è il seguente :

```
//oggetto di base
function Veicolo (marca,tipologia){
      this.marca = marca ||'sconosciuta';
      this.tipologia = tipologia || 'sconosciuta';
}
//oggetto derivato
function Auto (marca,targa){
      this.base = Veicolo; // richiama modello di base
      this.base(marca,'Automobile');
      this.targa = targa;
}
Auto.prototype =  new Veicolo; // associa il prototipo
//oggetto derivato a sua volta da Auto
function Fiat (targa){
      this.base = Auto;
      this.base('Fiat',targa);
}
Fiat.prototype =  new Auto; // associa il prototipo
//a questo punto se instanziamo Fiat con il parametro targa:
var a= new Fiat('CC893LJ');
//possiamo avere :
document.write(a.marca ); //'Fiat'
document.write(a.tipologia); //'Automobile'
document.write(a.targa); //'CC893LJ'
```

## 38 Implementare l'ereditarietà multipla tra gli oggetti

Quella che a prima vista sembrava una carenza di Javascript (la mancanza di classi) si rivela invece una straordinaria flessibilità se pen-

siamo alla straordinaria semplicità con cui è addirittura possibile implementare l'ereditarietà multipla tra gli oggetti (cioè quando, ad esempio, l'oggetto C eredita proprietà e metodi sia dall'oggetto A che dall'oggetto B). Prima di tutto estendiamo l'oggetto Object con prototype in modo da dotare tutti gli oggetti  del metodo extend che consente di passare i membri di un oggetto ad un altro oggetto :

```
Object.prototype.extend = function (source) {
 for (var member in source) {
      if(!(member in this)) { //evita di sovrascrivere membri esistenti
            this[member] = source[member];
      }
 }
}
```

Questa semplice funzione consente di assegnare tutti i membri di un oggetto (source) all'oggetto corrente (salvo il caso in cui l'oggetto corrente non disponga già di un membro con lo stesso nome). L'utilizzo è quanto mai facile :

```
var oggetto1 = new Object;
oggetto1.unaProprieta = 12;
var oggetto2 = new Object;
oggetto2.extend (oggetto1);
document.write(oggetto2.unaProprieta); //=12
```

Per applicare la tecnica in modo di realizzare l'eredità multipla pensiamo di avere un oggetto Indirizzo come :

```
//oggetto Indirizzo
function Indirizzo (via,comune,provincia) {
      this.via = via || '';
      this.comune = comune || '';
```

```
    this.provincia = provincia || '';
}
Indirizzo.prototype.indirizzoCompleto = function  (){
    return  this.via + " " + this.comune + " " + this.provincia;
}
```

Le proprietà e i metodi di Indirizzo possono essere comuni in molte situazioni : un indirizzo può essere associato ad esempio ad una Persona, o anche ad un Immobile. Definiamo quindi sia l'oggetto Persona che l'oggetto Immobile :

```
//oggetto Persona
function Persona (eta,genere,nome){
    this.eta = eta || 30;
    this.genere = genere || 'f';
    this.nome = nome || '';
}
Persona.prototype.maggiorenne = function  (){
    return (this.eta >= 18) ;
}
//oggetto Immobile
function  Immobile(mq,prezzo){
    this.mq = mq || 0;
    this.prezzo = prezzo || 0;
}
Immobile.prototype.valutaPrezzo = function (){ //metodo nel costruttore
    if (this.prezzo<=100000) return 'basso';
    else if (this.prezzo>100000 && this.prezzo<=300000) return 'medio';
    else return 'alto';
}
```

Adesso uniamo Persona e Indirizzo nell'oggetto Contatto, semplicemente con :

```
//oggetto  Contatto (Persona + Indirizzo)
function Contatto (eta,genere,nome,via,comune,provincia){
        this.extend(new Persona(eta,genere,nome));
        this.extend(new Indirizzo(via,comune,provincia));
}
```

Contatto quindi eredita sia da Persona che da Indirizzo, ovvero dispone di proprietà e metodi sia dell'uno che dell'altro oggetto. Tanto che è possibile scrivere :

```
var c = new Contatto(32,'m','Mario Rossi','via Verdi 3','Milano','MI');
document.write(c.nome); // Mario Rossi
document.write(c.indirizzoCompleto()); //via Verdi 3 Milano MI
```

Ma, come abbiamo detto, Indirizzo può essere riutilizzato anche in un contesto diverso, come ad esempio in un oggetto Appartamento, che eredita sia da Immobile che da Indirizzo :

```
//oggetto Appartamento(Immobile + Indirizzo)
function  Appartamento(mq,prezzo,via,comune,provincia){

        this.extend(new Immobile(mq,prezzo));
        this.extend(new Indirizzo(via,comune,provincia));
}
```

L'ereditarietà multipla è uno strumento tanto potente quanto pericoloso, tant'è vero che linguaggi come Java e C# la escludono. Il problema si ha infatti quando due classi (in questo caso sarebbe meglio dire due prototipi) hanno proprietà o metodi con lo stesso nome:

**ARTICOLO**
"nome
"prezzo

**CLIENTE**
“nome
“indirizzo

**ORDINE (ARTICOLO + CLIENTE)**
“nome ? (di ARTICOLO o di CLIENTE?)
“indirizzo
“prezzo

Si crea in questo caso un'ambiguità : la proprietà nome sarà ereditata da ARTICOLO o da CLIENTE? Nella funzione extend, che abbiamo visto sopra, la questione è stata risolta in modo un po' semplicistico evitando di ereditare un membro quando esso è già presente:

```
Object.prototype.extend = function (source) {
 for (var member in source) {
     if(!(member in this)) { //evita di sovrascrivere membri esistenti
          this[member] = source[member];

     }
 }
}
```

si potrebbe essere tentati invece di lanciare un'eccezione nel caso di membri già presenti:

```
Object.prototype.extend = function (source) {
 for (var member in source) {
     if((member in this)) throw member + " è già presente!";
     this[member] = source[member];

 }
}
```

ma questa strada si rivelerebbe impercorribile in quanto proprio la stessa funzione extend è stata dichiarata comune a tutti gli oggetti e quindi l'eccezione verrebbe sempre sollevata. Ci potrebbe anche essere la tecnica di "prefissare" i membri comuni con il nome del costruttore originario come in :

```
Object.prototype.multipleInherits = function (){
    var members = [];
    var args = arguments;
    var hasCollision = function  (memberName){
        var founds = 0;
        for(var i=0;i<args.length;i++){
            if (memberName in args[i]) founds++;
        }
        return (founds>1);
    }
    var getContructorName = function  (obj){
        try{
            return obj.constructor.toString().
                                match(/^function\s+(\w+)/)[1];
        }
        catch(e){
            return 'unknown';
        }
    }
    for(var i=0;i<args.length;i++){
     var source=args[i];
     for (var member in source) {
        if(!hasCollision(member)) {
            this[member] = source[member];
        }
        else {
            this[getContructorName(source) + '_' + member]
```

```
                                                                    = source[member];
            }
        }
    }
}
```

utilizzabile con :

```
function  Appartamento(mq,prezzo,via,comune,provincia){
    this.multipleInherits (

new Immobile(mq,prezzo),
new Indirizzo(via,comune,provincia)
);
}
```

In questo modo se,ad esempio, Immobile e Indirizzo, hanno tutti e due una proprietà che si chiama id, in Appartamento avremo Immobile_id e Indirizzo_id. Purtroppo anche questa soluzione non è affidabile al 100% perché se la proprietà, all'interno dell'oggetto base viene utilizzata ad esempio all'interno di un metodo, il metodo stesso non funzionerebbe all'interno dell'oggetto che eredita, cioè se all'interno di Immobile ci fosse:

```
Immobile.prototype.getId = function  (){
    return this.id  ;
}
```

Questo metodo all'interno di Appartamento perderebbe di significato in quanto la proprietà non sarebbe più id ma Immobile_id. In conclusione diciamo che è meglio lasciare all'implementatore dell'ereditarietà multipla l'onere di verificare che i nomi dei membri siano consistenti.

# OTTIMIZZAZIONE ED EFFICIENZA IN JAVASCRIPT

## 2.1 OTTIMIZZAZIONE DEL CODICE

### 39 Organizzare le strutture If-Else

In tutta la programmazione si fa spesso uso di strutture condizionali come If-Else. Per aumentare l'efficienza del codice (e quindi la velocità di esecuzione) è buona norma organizzare la struttura mettendo all'inizio le condizioni più comuni, in questo modo Javascript non dovrà valutare tutte le volte quelle meno comuni. Ad esempio si consideri questo codice:

```
if(mese =='febbraio' and giorno ==29) {
      eseguiCodice29Feb ();
}
else if(mese=='febbraio' and giorno!=29) {
      eseguiCodiceFeb ();
}
else {
      eseguiCodiceNormale ();
}
```

ebbene questo esempio è estremamente inefficiente perché per eseguire la condizione più comune deve comunque valutare anche le altre due, meglio sarebbe stato scrivere:

```
if(mese != 'febbraio') {
      eseguiCodiceNormale ();
}
else {
      if(giorno!=29) eseguiCodiceFeb ();
```

```
    else eseguiCodice29Feb ();
}
```

## 40 Uso di strutture Switch invece di If-Else

Quando si devono valutare numerose condizioni la struttura Switch è molto più efficiente (fino al doppio) di If-Else.
Il seguente codice:

```
    if(cond==1) ...
    else if(cond==2) ...
    else if(cond==3) ...
    else if(cond==4) ...
    else ...
```

sarebbe pertanto più efficiente se scritto come :

```
    switch(cond){
        case 1: ...
        case 2: ...
        case 3: ...
        case 4: ...
        default: ...
    }
```

## 41 Variabili non necessarie

La creazione di variabili in Javascript richiede tempo e memoria, molto spesso si creano variabili utilizzate solo una volta, come in:

```
var oggi = new Date();
var messaggio = "oggi è il giorno " + oggi.toString();
in modo più efficiente si sarebbe potuto scrivere:
var messaggio = "oggi è il giorno " + (new Date()).toString();
```

Come rovescio della medaglia c'è da dire però che la forma ottimizzata è spesso meno leggibile, ed anche la leggibilità del codice è un fattore da tenere presente.
Un buon compromesso è quello di evitare le variabili non necessarie soprattutto all'interno di cicli usati più volte.

# 2.2 EFFICIENZA REALE DEGLI SCRIPT IN AMBIENTE INTERNET

## 42 Limitare i commenti

Javascript come linguaggio interpretato e non compilato è più lento (si dice circa 5000 volte più lento del C, 100 di Java e 10 di Perl) tuttavia poiché si parla in termini di millisecondi, o anche meno, la differenza non si nota più di tanto, considerando anche i moderni processori e la dotazione hardware dei client; il problema in realtà è un altro: non il tempo in cui lo script viene eseguito, ma quello in cui viene scaricato dal sito. Consideriamo infatti che nel processo di acquisizione di un file da Internet abbiamo diversi "colli di bottiglia" : la velocità con cui il client può scaricare il file, la velocità con cui il server può trasmettere il file ecc... Da diversi test, effettuati in diverse condizioni, risulta che un file con una normale linea ADSL si scarica mediamente in circa 18 ms per ogni Kb (è naturalmente una media, le condizioni sono le più varie). Se pensiamo che pagine ricche di logica Javascript possono tranquillamente avere anche 200-300 Kb di codice da scaricare ne ricaviamo che questo codice prima di essere compilato ed eseguito per la prima volta (le volte successive probabilmente è in cache del browser) c'è un tempo di attesa dell'utente di 6/10 secondi (e questo solo per gli script, senza parlare delle immagini, CSS ecc...). Quindi serve a poco guadagnare qualche manciata di millisecondi in fase di esecuzione se poi il nostro script impiega 1 o 2 secondi in più per essere scaricato! Da questo punto di vista ogni Byte risparmiato è tempo guadagnato! Una cosa che po-

trebbe sembrare ovvia (ma non lo è) è che anche i commenti al codice, utilissimi in fase di sviluppo e di mantenimento, rappresentano Kb in più da scaricare e quindi in fase di produzione è necessario eliminarli o limitarli il più possibile. L'ideale sarebbe realizzare due versioni dei nostri scripts, una commentata per lo sviluppo ed una senza commenti per la produzione. In questo modo si arriva a risparmiare anche il 10-20% di spazio occupato dai files.

## 43 Limitare le inclusioni di file esterni

Il meccanismo richiesta/risposta tra client e server prevede grosso modo tre fasi:

- il client invia un header (request header) contenente il nome del file da scaricare
- il server invia un altro header (response header) contenente il risultato della richiesta (il file esiste o non esiste, c'è stato o no un errore ecc...)
- il server, in caso in cui accetti la richiesta, invia il file vero e proprio

Questo meccanismo avviene per ogni file presente nella pagina (Html, Javascript, CSS, immagini ecc...). Programmando in javascript noi abbiamo la possibilità di includere dei file esterni senza dover ogni volta scrivere il codice nella pagina :

```
<script src="fileEsterno.js" type="text/javascript"></script>
```

Questo ci permette di riutilizzare uno stesso script per più pagine e dividere per "categorie di funzioni" gli script da includere, sono le c.d. librerie Javascript. Dobbiamo però tener presente che ognuno di questi file deve essere scaricato con quelle 3 fasi che dicevamo prima. A parità di quantità di codice risulta quindi molto più efficiente scaricare pochi grossi file che molti piccoli file anche se la somma to-

tale in Kb è la stessa. Molto spesso l'inclusione di file esterni è comunque inevitabile, tuttavia altrettanto spesso ci facciamo poco caso e facendo il copia e incolla degli script di inclusione tra una pagina e l'altra si finisce per includere anche script che nella pagina corrente non verranno mai utilizzati.

## 44 Scriversi proprie librerie

Grazie ad Internet (e a Google!) il web è una vera e propria miniera di librerie Javascript per tutte le esigenze: dalla gestione del DOM, ai grafici, a controlli personalizzati ecc... Il problema è che chi scrive una libreria non può sapere quali altri script includerà il programmatore nella pagina. Prendiamo un caso frequente : sapere con quale browser si sta visualizzando la pagina. Lo sviluppatore di una libreria ha spesso bisogno di sapere il browser in uso per scrivere funzioni differenti a seconda del client, questo comporta che ogni sviluppatore inserirà nella libreria le proprie funzioni di riconoscimento. Il risultato sarà che se il programmatore include 3 librerie Javascript che pur svolgono compiti diversi tutte e 3 le librerie avranno dentro una buona quota di funzioni che fanno la stessa cosa. Mettiamo, semplicisticamente, che le funzioni comuni siano il 20% e i file "pesino" ognuno 100Kb, il risultato sarà che in 300Kb abbiamo ben 40 Kb di funzioni duplicate! Tutto questo ovviamente non vuol dire che non dobbiamo più scaricare le librerie Javascript dal Web, oltretutto analizzare il codice dei programmatori più bravi è un ottimo metodo per imparare, vogliamo dire solo che con il tempo e un po' di esperienza dovremmo cercare di farci le nostre librerie personalizzate, magari anche mettendo insieme parti di codice prese qua e là. Questo ci porterà ad ottimizzare il codice che inseriamo nelle nostre pagine ed anche (il che non guasta mai) a capire cosa stiamo usando.

# NUOVA GESTIONE DEL DOM HTML E STILI

## 3.1. GESTIONE DEGLI EVENTI

La manipolazione del DOM HTML è una delle funzionalità più importanti dell'uso di Javascript nell'ambito web. Il DOM espone a Javascript gli elementi, gli attributi ed il testo stesso contenuti nella pagina web sotto forma di oggetti, che possono essere anche modificati, creati o cancellati programmaticamente. Molti oggetti poi dispongono di eventi, l'evento non è altro che un messaggio, che il browser invia a Javascript; ad esempio per l'elemento DIV è definito anche l'evento onclick che si verifica quando l'utente fa click con il tasto sinistro del mouse sull'elemento, in questo caso il browser notifica il messaggio a Javascript e, se il programmatore ha associato all'evento una funzione, tale funzione verrà eseguita.

### 45 Ottenere i riferimenti agli elementi del DOM

Il primo problema da risolvere è proprio quello di come fare ad ottenere i riferimenti agli oggetti esposti dal DOM. Prendiamo a riferimento dei nostri esempi una breve pagina HTML :

```
<html>
<head>
	<title>DOM HTML</title>
</head>
<body>
	<div id="elm1">primo elemento</div>
	<div id="elm2">secondo elemento
		<div>DIV nidificato</div>
	</div>
</body>
</html>
```

La pagina stessa è rappresentata dall'oggetto document. Per ottenere i riferimenti agli elementi abbiamo due metodi, uno proprio di document, getElementById, ed un altro comune a tutti gli elementi (document compreso), getElementsByTagName. Cominciamo proprio da quest'ultimo; getElementsByTagName, come suggerisce il nome, serve per ottenere una lista di elementi a partire dal nome dell'elemento. Per ottenere, nel nostro caso, tutti gli elementi DIV dovremmo quindi usare:

```
window.onload = function () {
    var divs = document.getElementsByTagName('DIV');
    alert(divs.length); // risultato 3
}
```

La funzione restituisce, in forma di lista, i riferimenti a tutti gli elementi presenti sotto l'elemento di partenza indipendentemente dalla loro posizione. Se volessimo invece ottenere una lista di tutti i riferimenti agli elementi HTML sottostanti all'elemento di partenza basta sostituire il nome del tag con un asterisco :

```
var tutti = document.getElementsByTagName('*');
```

getElementsByTagName talvolta è utile, ma ancor più preziosa è la funzione getElementById che consente di ottenere il riferimento ad un oggetto a partire dall'attributo id in esso definito. Nel nostro caso per ottenere il riferimento al primo elemento DIV possiamo usare:

```
var div1 = document.getElementById('elm1');
```

nel caso in cui nel documento vi siano due elementi con lo stesso id la funzione restituisce solo il primo. Internet Explorer consente anche di ottenere il riferimento all'elemento richiamando direttamente il nome :

```
var testo = elm1.innerHTML;
```

anziché

```
var testo = document.getElementById('elm1').innerHTML;
```

questo sistema potrebbe essere "attraente", visto che consente di risparmiare un bel po' di battute in fase di programmazione ed anche, perché no, anche un bel po' di bytes al nostro codice, tuttavia esso non è conforme allo standard del W3C. È vero che gli altri browser come Opera e Firefox si sono, per così dire "rassegnati" a questo andazzo e quindi non generano errore per questo tipo di chiamate (Firefox si limita a segnalarle come Warning in fase di debug) però non è sempre meglio non deviare dagli standard a meno di non avere una buona ragione.

## 46 Scorciatoie per i metodi del DOM

La funzione document.getElementById è un po' come il prezzemolo, viene utilizzata dappertutto e quindi doverla digitare ogni volta è una sofferenza! Si può comunque ovviare al problema, in maniera più elegante, semplicemente "wrappando" la funzione così :

```
function $ (e){
    if(typeof e =="string") return document.getElementById(e);
    else if(typeof e =="object") return e;
    else return null;
}
```

Il carattere $ non è infatti tra quelli proibiti per i nomi di oggetti in Javascript e si presta bene ad essere utilizzato come scorciatoia facilmente memorizzabile. Dichiarando questa funzione d'ora in poi nel nostro codice potremmo tranquillamente scrivere:

```
var testo = $('elm1').innerHTML;
```

anziché

```
var testo = document.getElementById('elm1').innerHTML;
```

rispettando tutti gli standards del caso.

## 47 Gestire l'evento Onload

Negli esempi precedenti abbiamo inserito il nostro codice in una funzione assegnata all'evento onload dell'oggetto window. Premesso che vedremo meglio più avanti (vedi punto 75) cosa sono gli eventi , non è possibile non accennare adesso al significato di onload. Finchè la pagina Web non è del tutto interpretata dal browser, gli elementi del DOM non sono disponibili a Javascript. Se noi scrivessimo del codice come :

```
<html>
<head>
    <script language="javascript" type="text/javascript">
        var div1 = document.getElementById('elm1');
        alert(div1.innerHTML);
</script>
</head>
<body>
    <div id="elm1">primo elemento</div>
…
</body>
</html>
```

Andremmo incontro inevitabilmente ad un errore perché lo script andrebbe eseguito quando ancora l'elemento DIV non è stato letto dal browser (la lettura avviene in sequenza dall'altro verso il basso). onload invece viene notificato proprio quando il browser ha terminato di leggere il codice HTML e quindi a quel punto tutti gli elementi del DOM sono certamente disponibili. Il codice che fa riferimento al DOM deve quindi es-

sere dichiarato all'interno del gestore dell'evento onload con uno dei metodi che vedremo successivamente, come ad esempio:

```
<html>
<head>
    window.onload = function () {
        //esegui qui il codice DOM
    }
</head>
...
```

A dire il vero si potrebbe anche dichiarare il codice DOM dopo che nell'HTML sono stati definiti gli elementi, perché quando viene valutato lo script essi sono già presenti:

```
<html>
<head>
</head>
<body>
    <div id="elm1">primo elemento</div>
    <script language="javascript" type="text/javascript">
        var div1 = document.getElementById('elm1');
        alert(div1.innerHTML);
</script>
…

</body>
</html>
```

perché nel momento in cui è dichiarato l'elemento è già stato letto. Tuttavia si ricorre a questo tipo di dichiarazione più raramente, perché tende a “mischiare” troppo la logica Javascript con la struttura HTML.

# 48 Proprietà e metodi degli oggetti HTML del DOM

Una volta ottenuto il riferimento all'oggetto DOM è possibile leggere e impostare le proprietà relative e richiamare i metodi. In genere il nome delle proprietà è lo stesso degli attributi HTML per cui, ad esempio, in un oggetto che si riferisce ad un elemento TABLE per ottenere la larghezza potremo usare:

```
var tableRef = document.getElementById('myTable');
var w = tableRef.width ;
così come naturalmente potremo impostare anche il valore con:
tableRef.width = "100%";
```

Alcune proprietà però hanno un nome diverso dall'attributo HTML, tra queste particolarmente importante è className, la proprietà che restituisce o imposta la classe associata all'elemento. Essa dovrà quindi essere usata così:

```
var myElement = document.getElementById('myElement');
myElement.className="regola";
```

Nell'HTML moderno però molte proprietà di formattazione sono in realtà impostate attraverso gli stili CSS, style è quindi una proprietà disponibile per gli elementi del DOM, tuttavia esso è a sua volta un oggetto con proprietà che ricalcano le regole impostabili attraverso i CSS. Ovvero non è possibile, ad esempio, impostare i bordi di un elemento utilizzando:

```
var myElement = document.getElementById('myElement');
myElement.style="border:1px solid red";
```

dovremmo invece utilizzare:

```
myElement.style.border = "1px solid red";
```

## 49 Proprietà comuni del DOM

Oltre alle proprietà "tipiche" dei vari tag HTML il DOM espone delle proprietà comuni a tutti i nodi. Precisiamo che per Nodo si intende qualsiasi entità significativa per HTML, analizziamo questo frammento HTML:

```
<div id="2">testo</div>
```

abbiamo:

- nodo di nome div di tipo elemento
- nodo di nome id di tipo attributo con valore 2 figlio del nodo div
- nodo di testo di tipo testo con valore "testo" figlio del nodo div

Ogni nodo ha come proprietà :

- nodeName - nome del nodo, ad esempio "div" per l'elemento e "id" per l'attributo, per i nodi di testo il nome invece è sempre "#text"
- nodeType - codice che rappresenta il tipo : 1 per gli elementi, 2 per gli attributi, 3 per i nodi di testo (ci sono anche altri tipi di nodo, ma sono meno usati).
- nodeValue - valore del nodo che restituisce il testo nel caso di attributi e nodi di testo è null per gli elementi.
- childNodes - lista degli elementi e dei nodi di testo contenuti in un nodo, ovviamente questo ha senso solo per gli elementi, gli attributi e i nodi di testo non contengono nodi.
- parentNode - nodo di livello superiore, per il nodo di primo livello parentNode è null.

Ad esempio, per ricavare il nome del nodo:

```
var oDiv = document.getElementById("myDiv");
alert(oDiv.nodeName);
```

## 50 Creazione di elementi

Una delle funzioni più utili del DOM è la creazione dinamica dell'HTML, intendendosi con creazione dinamica l'aggiunta di nuovi nodi (elementi, attributi nodi di testo ecc…) al codice HTML originariamente presente nella pagina. Prima di aggiungere un nuovo elemento però occorre crearlo, trattandosi di un elemento occorre richiamare il metodo createElement dell'oggetto document.

```
var oDiv = document.createElement('div');
oDiv.align = "right";
//ecc...
```

## 51 Creazione di nodi di testo

Analogamente agli elementi anche per inserire dinamicamente del testo si utilizza un metodo dell'oggetto document , il metodo createTextNode, utilizzando come argomento la stringa da inserire:
var oText = document.createTextNode("testo da inserire");

## 52 Creazione e lettura di attributi

L'oggetto document dispone anche del metodo createAttribute, tuttavia per creare (e nello stesso tempo aggiungere) un attributo è estremamente più comodo utilizzare setAttribute che è tipico di ogni nodo elemento. Ad esempio per aggiungere un attributo ad un DIV è possibile semplicemente:

```
var oDiv = document.createElement('div');
oDiv.setAttribute("id",2);
```

Allo stesso modo si utilizza getAttribute per leggere il valore di un attributo di un elemento:

```
var oDiv = document.getElementById("myDiv");
var id = oDiv.getAttribute("id");
```

## 53 Aggiunta di nodi al documento

Per gli attributi, come abbiamo visto, utilizzare setAttribute significa non solo creare, ma anche aggiungere il nodo all'elemento; nel caso di nodi di testo ed elementi invece ciò non avviene automaticamente. Se la creazione del nodo avviene utilizzando createElement e createTextNode di document, l'aggiunta deve avvenire ad un elemento specifico ecco perché ogni nodo di tipo elemento dispone di alcuni metodi per aggiungere sotto di sé altri nodi:

- appendChild - aggiunge il nodo in coda agli altri
- insertBefore - aggiunge il nodo prima di un altro

Ad esempio, dato un elemento SELECT:

```
<select id="mySel"></select>
```

è possibile aggiungere dinamicamente delle OPTION con il DOM così:

```
var oSel = document.getElementById('mySel');
var option1 = document.createElement('option');
option1.value = '1';
option1.appendChild(document.createTextNode('opzione 1'));
var option2 = document.createElement('option');
option2.value = '2';
option2.appendChild(document.createTextNode('opzione 2'));
oSel.appendChild(option1);
oSel.appendChild(option2);
```

il risultato sarà:

```
<select id="mySel">
<option value="1">opzione 1</option>
<option value="2">opzione 2</option>
</select>
```

## 54 Rimuovere i nodi

Così come è possibile aggiungere elementi e altri tipi di nodo attraverso il DOM è ovviamente possibile anche rimuovere dinamicamente nodi esistenti. L'operazione va effettuata richiamando il metodo removeChild dell'elemento superiore a quello da rimuovere passando, come parametro, il riferimento al nodo da rimuovere. È appena il caso di ricordare che l'elemento superiore a quello corrente è sempre ricavabile dalla proprietà parentNode, quindi per rimuovere un determinato elemento basterà:

```
var oDiv = document.getElementById('myDiv');
oDiv.parentNode.removeChild(oDiv);
La rimozione dei nodi di testo funziona allo stesso modo:
var oDiv = document.getElementById('myDiv');
      var oText = oDiv.childNodes[0]; //se il nodo di testo
  //è il primo nodo figlio
oDiv.removeChild(oText);
```

anche se in questi casi non è infrequente trovare il tanto vituperato (vedi punto 58) :

```
oDiv.innerHTML=" ";
```

la rimozione degli attributi in genere serve a poco, comunque si ottiene attraverso il metodo removeNamedItem della collection attributes di un elemento:

```
var oDiv = document.getElementById('myDiv');
oDiv.attributes.removeNamedItem("id");
```

## 55 Scambiare la posizione di due elementi

Il DOM di Internet Explorer dispone di una comoda funzione per scambiare la posizione di due elementi, swapNode:

```
element1.swapNode(element2);
//scambia la posizione di element1 con quella di element2
che, usando i metodi standard, equivale a:
var nextSibling = element1.nextSibling;
var parentNode = element1.parentNode;
if(element2.parentNode) node.parentNode.replaceChild(element1,
                                                      element2);
parentNode.insertBefore(element2, nextSibling);
```

Se non vogliamo rinunciare alla comodità di swapNode anche in Firefox possiamo comunque estendere il prototipo di Node:

```
try{
    Node.prototype.swapNode = function (node) {
        var nextSibling = this.nextSibling;
        var parentNode = this.parentNode;
          if(node.parentNode) node.parentNode.replaceChild(this, node);
        parentNode.insertBefore(node, nextSibling);
    };
}
catch(e){}
```

l'uso di try-catch ci garantisce in questo caso che la funzione non generi errore in IE.

## 56 Trovare un elemento superiore

Usando un ciclo while e sfruttando la proprietà parentNode di un elemento è facilmente possibile ottenere il riferimento a un elemento superiore. Ad esempio per trovare il nodo superiore all'elemento (e) che abbia un certo nome (nodeName):

```
function findAncestor (e,nodeName){
    //trova un elemento superiore
```

```
    nodeName = nodeName.toLowerCase()
    while (e.nodeType == 1 &&  e.nodeName.toLowerCase()
                                                !=nodeName) {
        e = e.parentNode;
    }
    return e;
}
//uso:
var node = findAncestor(myElement,'div');
```

## 57 Trovare un elemento allo stesso livello

Sempre attraverso un ciclo while e sfruttando le proprietà nextSibling (nodo successivo allo stesso livello) e previousSibling (nodo precedente allo stesso livello) di un elemento è possibile ottenere il riferimento a un elemento di pari livello. Ad esempio per trovare il nodo di pari livello all'elemento (e) che abbia un certo nome (nodeName) e sia collocato prima o dopo (direction):

```
function findSibling (e,nodeName,direction) {
    var sibling = (direction=='next')?'nextSibling':'previousSibling';
    nodeName = nodeName.toLowerCase()
    while(e!=null && e.nodeName.toLowerCase()!=nodeName){
        e = e[sibling];
    }
    return e;
}
//uso:
var node = findAncestor(myElement,'li','previous');
var node = findAncestor(myElement,'li','next');
```

## 58 innerHTML e innerText

Internet Explorer, come avremo più volte modo di vedere, adotta un'implementazione per così dire "estensiva" del DOM codificato dal W3C; "esten-

siva" nel senso che le cose prescritte dallo standard (più o meno) ci sono, ma ci sono anche tante altre funzioni proprietarie. Tra esse spiccano le proprietà innerHTML e innerText, la prima consente di aggiungere dinamicamente frammenti di codice HTML in forma di stringa, la seconda invece permette di impostare direttamente il nodo di testo di un elemento. Ad esempio, volendo creare dinamicamente questo frammento HTML:

```
<div>
    <span id="span1">testo</span>
</div>
usando lo standard W3C scriveremmo:
    var oDiv = document.createElement('div');
    var oSpan = document.createElement('span');
    oSpan.id = "span1";
    oSpan.appendChild(document.createTextNode('testo'));
    oDiv.appendChild(oSpan);
    document.body.appendChild(oDiv);
```

mentre con innerHTML sarebbe sufficiente:

```
    var oDiv = document.createElement('div');
    oDiv.innerHTML='<span id="span1">testo</span>';
    document.body.appendChild(oDiv);
```

Se invece dovessimo aggiungere del testo a un elemento come:

```
<div id="message"></div>
```

usando lo standard W3C scriveremmo:

```
    var oDiv = document.getElementById('message');
    var text = document.createTextNode('testo messaggio');
    oDiv.appendChild(text);
```

mentre con innerText sarebbe sufficiente:

```
var oDiv = document.getElementById('message');
    oDiv.innerText='testo messaggio';
```

Di fronte a questa "semplificazione" molti a dir la verità sono tentati di seguire queste scorciatoie e anche gli altri browser come Firefox e Opera, per non perdere una platea di sviluppatori, sono stati costretti ad inserire nel loro DOM anche innerHTML e innerText (Firefox soltanto innerHTML) pur sconsigliandone l'uso. C'è da dire però che anche se innerHTML ormai funziona con la maggior parte dei browser in circolazione (non altrettanto innerText che però è in qualche modo ricompreso in innerHTML), il suo uso è da prevedersi con parsimonia:

- Prima di tutto ci sono dei casi in cui non funziona, come ad esempio se volessimo aggiungere delle OPTION a una SELECT:

```
var oSel = document.getElementById('mySel');
var sHTML = '<option value="3">opzione 3</option>';
sHTML += '<option value="4">opzione 4</option>';
oSel.innerHTML += sHTML;
```

  anzi c'è da dire che non funziona proprio in IE, mentre non dà problemi in Firefox!

- Il secondo motivo è che si perde la possibilità di estendere gli elementi con proprietà e metodi personalizzati, se con il DOM standard  posso scrivere

```
    var oDiv = document.createElement('div');
    var oSpan = document.createElement('span');
    oSpan.id = "span1";
oSpan.myCustomProperty = "3";
```

```
oSpan.getMyCustomProperty = function (){return this.myCustomProperty}
oSpan.onclick = function () {alert(this.getMyCustomProperty())}
    oSpan.appendChild(document.createTextNode('testo'));
    oDiv.appendChild(oSpan);
    document.body.appendChild(oDiv);
```

con innerHTML non avrei questa possibilità

- Il terzo motivo è forse più estetico; inserire chilometri di stringhe frammiste al codice non è certo la cosa più bella da vedere (e da mantenere) e ci porta al famigerato "spaghetti code" che da sempre affligge alcune tecniche di programmazione come ASP.

Detto questo però non vorremmo far la parte dei "pasdaran" dello "o standard o morte", innerHTML ormai c'è ed è uno standard de facto, quindi facciamone un uso parco e consapevole:

- innerHTML è comunque un metodo rapido per cancellare in un colpo solo il contenuto di un elemento, sottoelementi compresi

```
    var oDiv = document.getElementById('myDiv');
oDiv.innerHTML=""; //ora oDiv è vuoto
```

- innerHTML non presenta controindicazioni se usato per impostare i nodi di testo (può compiere tranquillamente la stessa funzione di innerText, meno supportato)

```
    var oDiv = document.getElementById('myDiv');
    oDiv.innerHTML="testo messaggio";
```

- Al limite è anche comprensibile l'uso di innerHTML per inserire degli elementi fissi come le immagini che con il DOM richiederebbero diverse righe di codice

```
    var oDiv = document.getElementById('myDiv');
    oDiv.innerHTML="<img src='picture.gif' border='0' title=
                                                    'my picture'/>";
```

## 59 Aggiungere funzionalità agli oggetti DOM

Gli oggetti del DOM, come qualsiasi altro oggetto in Javascript, sono espandibili nel senso che in qualsiasi momento si possono aggiungere, per semplice assegnazione, nuove proprietà e metodi. Se ad esempio ad un oggetto riferito a un DIV volessimo aggiungere un metodo che ne recupera le dimensioni potremmo tranquillamente scrivere:

```
var oDiv = document.getElementById('myDiv');
oDiv.getSize = function() {
    return {width:this.offsetWidth,height: this.offsetHeight}
}
```

per poi riutilizzarlo quando occorre:

```
    var size = oDiv.getSize();
    alert("Larghezza:" + size.width + " Altezza:" + size.height);
```

## 60 Prototipi degli oggetti DOM

Alcuni browser come Firefox offrono un'opportunità ancora più ampia rispetto all'estensione del singolo oggetto DOM : l'accesso ai prototipi degli oggetti stessi. Ciò significa che potremmo dotare ogni elemento di una certa funzione. Proseguendo con l'esempio fatto al punto 59 potremmo aggiungere il metodo getSize a tutti gli elementi referenziati con il DOM semplicemente con:

```
Element.prototype.getSize = function() {
    return {width:this.offsetWidth,height: this.offsetHeight}

}
```

Questa tecnica è molto elegante e flessibile ed è un vero peccato che non sia stata adottata anche da Internet Explorer. Comunque non è del tutto priva di impieghi pratici, infatti viene comunemente usata per mimare (o come si dice in gergo, wrappare) metodi e proprietà tipici del DOM di Internet Explorer anche in Firefox. Ne abbiamo un esempio al punto 75 dove abbiamo introdotto nel DOM di

# 3.2 FIREFOX IL METODO ATTACHEVENT PROPRIO DI IE

## 61 Scorrere l'albero del DOM

Il DOM, come abbiamo visto, presenta una struttura ad albero, dove ogni elemento (eccetto quelli di primo livello) presenta un elemento superiore (parentNode) e n elementi o nodi di testo figli (childNodes), in questa struttura lo scorrimento può avvenire dalla radice alle foglie (cioè dagli elementi di primo livello a quelli più profondi) :

```
function walkNodes (currentNode) {
    // usa currentNode per operazioni
    for(var i=0;i<currentNode.childNodes; i++) {
        walkNodes(currentNode.childNodes[i]); //funzione ricorsiva
}
}
```

ma possiamo avere anche uno scorrimento da una foglia alla radice (da un elemento più profondo fino al primo livello):

```
currentNode = document.getElementById("myNode");
while(currentNode) {
    // usa currentNode per operazioni
currentNode = currentNode.parentNode;
}
```

esiste poi anche la possibilità di scorrere gli elementi dello stesso livello mediante le proprietà previousSibling e nextSibling che rappresentano gli elementi precedenti o successivi dello stesso livello:

```
currentNode = document.getElementById("myNode");
while(currentNode) {
	// usa currentNode per operazioni
currentNode = currentNode.previousSibling;
}
currentNode = document.getElementById("myNode");
while(currentNode) {
	// usa currentNode per operazioni
currentNode = currentNode.nextSibling;
}
```

## 62 Ricerca con XPath

Come abbiamo visto nel punto 61 attraverso la gerarchia dei nodi possiamo, attraverso dei cicli, scandire tutto l'albero del DOM alla ricerca dei nodi. Per esempio, potremmo voler cercare nel documento HTML un elemento con un attributo avente un certo valore, e quindi :

```
function getVisible (currentNode) {
	if(currentNode.getAttribute("visible")=="yes") {
		return currentNode;
}
	for(var i=0;i<currentNode.childNodes; i++) {
		walkNodes(currentNode.childNodes[i]); //funzione ricorsiva
}
}
```

La cosa non è difficile, ma indubbiamente un po' ripetitiva. In XML per la ricerca sui nodi esiste il linguaggio XPath che consente di risolvere una ricerca come quella che abbiamo visto semplicemente con:

```
var nodeList = document.evaluate("//*[@visible='yes']",[...]);
```

In realtà XPath è implementato anche per l'HTML : Firefox supporta XPath 3.0, Opera XPath 1.0 il problema però è che Internet Explorer non fornisce nessun supporto in questo senso, comunque ci sono alcune librerie (una di Cameron McCormack su http://mcc.id.au/xpathjs molto completa ma un po' pesante, un'altra su http://js-xpath.sourceforge.net, più leggera ma che non funziona con Opera) che consentono l'uso di XPath con anche con IE. Ecco un esempio della ricerca di tutti i tag A (i link) con l'attributo class uguale a "evidenza" con XPath 3.0 :

```
var xPathExpression = "//a[@class='evidenza']";
var result = document.evaluate
            (xPathExpression,document,null,XPathResult.ANY_TYPE, null);
var el = result.iterateNext();
while(el) {
    el.style.border = "2px solid red"; //aggiunge bordo rosso al link
    el=result.iterateNext();
}
```

Effettivamente si tratta di uno strumento molto potente, tuttavia il mancato supporto di IE (se non attraverso librerie aggiuntive) ne limita fortemente l'uso.

## 63 Mostrare e nascondere un elemento usando l'oggetto style

L' oggetto style associato agli elementi DOM consente di accedere attraverso Javascript alle proprietà CSS degli elementi, tra queste particolarmente utili sono quelle che regolano la visualizzazione di un elemento nella pagina, ovvero:

- **display** - "none" per nascondere l'elemento e "block","inline" o anche stringa vuota "" per mostrarlo

- **visibility** - "hidden" per nascondere l'elemento e "visible" per mostrarlo

Quindi per nascondere e mostrare un elemento da Javascript:

```
//mostra
var myEl = document.getElementById("myEl");
myEl.style.display = "none";
//nasconde
myEl.style.display = "";
oppure:
//mostra
myEl.style.visibility = "visible";
//nasconde
myEl.style.visibility = "hidden";
```

display e visibility non sono proprio la stessa cosa: se nascondo un elemento con display="none" lo spazio che prima occupava nella pagina verrà occupato dagli altri elementi, con visibility invece resta lo spazio vuoto, ragion per cui si tende ad usare visibility per gli elementi posizionati in modo assoluto (position="absolute") e display per gli altri.

## 64 Differenze tra nomi di stili CSS e proprietà style in elementi DOM

Da quello che abbiamo visto è facile intuire che le proprietà CSS tendono a corrispondere a corrispondenti proprietà nell'oggetto style associato agli elementi del DOM. Ad esempio, se in CSS possiamo impostare la larghezza di un elemento con:

```
<div style="width:100px">
```

Nel corrispondente oggetto DOM style ritroveremo un'analoga proprietà width:

```
oDiv.style.width="100px";
```

È chiaro che si è cercato di mantenere uniformità di denominazione tra CSS e DOM per facilitare l'uso degli stili attraverso il DOM da parte di chi abbia competenze su CSS. Tuttavia una stretta corrispondenza tra CSS e DOM non è del tutto possibile, ci sono infatti in CSS molte proprietà dal nome "composto" ovvero separato da trattini "-" come:

```
margin-top, text-align, background-color ecc…
```

Si è quindi introdotta la regola che, per questi nomi, come proprietà dell'oggetto style il trattino sparisce e la parola successiva ad esso assume l'iniziale maiuscola (lower Camel Case) e quindi:

```
marginTop, textAlign, backgroundColor ecc…
```

## 65 funzione per impostare gli stili in modo CSS-like

L'uniformità delle regole di denominazione delle proprietà CSS in style degli elementi del DOM ci consente di elaborare una funzione molto utile che ci consente di impostare style nello stesso modo con cui faremmo in HTML. La funzione trasforma una stringa con sintassi CSS in proprietà di style di un elemento:

```
    function setStyle (e,strStyle){
        if(e instanceof Array){
var i = e.length;
while(--i>=0){this.style(e[i],strStyle)};
return ;
}
        //applica style all'oggetto
        var getStylePropName = function (name){
            /*trasforma un nome di stile tipo "border-color"
```

```
                                        in camelCase "borderColor"*/
            var ss = name.split("-");var StylePropName="";
            for(var i=0;i<ss.length;i++){
                var chunk = ss[i].toLower();
                if (i>0){
var firstChar = chunk.charAt(0);
var restChars = chunk.substr(1);
chunk=firstChar.toUpperCase() + restChars.toLower();
}
                StylePropName+=chunk;
            }
            return StylePropName;
        }//getStylePropName
        var stylePairs = strStyle.split(";");
        for(var i=0;i<stylePairs.length;i++){
            var stylePair = stylePairs[i].split(":");
            var styleName = getStylePropName(stylePair[0])
            var styleValue = (stylePair[1])?stylePair[1]:"";
        //$(e) usa la funzione $ come scorciatoia per
        //document.getElementById come abbiamo visto al punto 46
            $(e).style[styleName]=styleValue;
        }
    }
```

La funzione si presta ad un uso multiforme.
Nella forma più semplice applica ad un oggetto DOM gli stili attraverso una sintassi CSS:

```
set Style(oDiv,"width:100px;height:10px;background-color:yellow");
risparmiando così di dover scrivere:
oDiv.style.width = "100px";
oDiv.style.height = "10px";
oDiv.style. backgroundColor = "yellow";
```

```
oltre che a passare come riferimento l'oggetto possiamo passare
                                                    anche solo l'id:
set Style("myDiv","width:100px;height:10px;background-color:yellow");
```

Ma la funzione fa di più; consente di applicare, in un colpo solo, lo stesso stile ad un Array di elementi (o di ID di riferimento):

```
set Style(["div1","div2","div3"],"width:100px;height:10px;
                                          background-color:yellow");
```

# 3.3 POSIZIONE E DIMENSIONI DEGLI ELEMENTI

## 66 Ottenere la posizione di un elemento

Una delle operazioni più "insidiose" è quella di ottenere la posizione corrente di un elemento all'interno della pagina. Le proprietà che registrano la distanza di un elemento dal margine sinistro e dal margine superiore sono , rispettivamente, offsetLeft e offsetTop. Tuttavia è molto meno chiaro cos'è che viene considerato "margine sinistro" e "margine superiore". Si consideri di dover misurare la posizione della cella bordata di rosso della **figura 66-1**. la cella in questione è contenuta in una tabella nidificata e centrata:

```
<table width="100%" border="1" cellpadding="3" cellspacing="4">
    <tr>
        <td align="center">
        <table cellpadding="2" border="1" cellspacing="2">
            <tr>
                <td>
                cella 1
                </td>
                <td id="cell" onclick="findPos('cell')">
```

**Figura 66-1:** Cella nidificata

```
                    trova la posizione di questa cella
                </td>
            </tr>
        </table>
        </td>
    </tr>
</table>
<div id="info"></div>
```

Vogliamo quindi sviluppare una funzione findPos che fa spostare il DIV con id 'info' posizionato proprio sull'offsetLeft e l'offsetTop della cella in questione, in questo DIV scriveremo proprio le misure trovate:

```
  function findPos (id){
        var e = document.getElementById(id);
        var info = document.getElementById('info');
        with(info.style) {
            position='absolute';
            background='#EEEEDD';
            border='1px solid #000';
            left= e.offsetLeft + "px";
```

```
                top =  e.offsetTop + "px";
            }
            info.innerHTML= e.offsetLeft + "px " +  e.offsetTop + "px ";
    }
```

il risultato darà però un effetto ben diverso da quello sperato (**figura 66-2**):

**Figura 66-2:** Posizione errata

in realtà ciò avviene perché, come evidenziamo in **figura 66-3**, vengono considerati come margini quelli della tabella che contiene la cella, non la pagina intera.
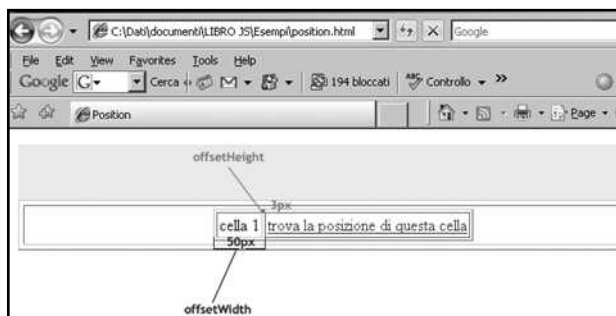
**Figura 66-3:** Margini

Per risolvere il problema occorre risalire la gerarchia dei nodi attraverso offsetParent (che restituisce l'elemento superiore che ha un offset, notare che è diverso da parentNode perché restituisce solo gli elementi che influiscono sull'offset) e sommare le varie distanze, per far questo utilizziamo la funzione:

```
function getOffset(element) {
   var valueT = 0, valueL = 0;
      do {
        valueT += element.offsetTop  || 0;
        valueL += element.offsetLeft || 0;
        element = element.offsetParent;
      } while (element);
   return {left:valueL, top:valueT};
}
```

modifichiamo quindi anche findPos:

```
function findPos (id){
     var e = document.getElementById(id);
     var info = document.getElementById('info');
     var offset = getOffset(e);

     with(info.style) {
          position='absolute';
          background='#EEEEDD';
          border='1px solid #000';
          left= offset.left + "px";
          top =  offset.top + "px";
     }

     info.innerHTML= offset.left + "px " +  offset.top + "px ";
}
```

Il risultato (come vediamo in figura 66-4) risulta ora corretto in quanto il DIV si posiziona esattamente tra l'inizio del margine sinistro e l'inizio del margine superiore della cella.

## 67 Dimensioni di un elemento

Un'altra importante operazione con cui ci si trova spesso ha che fare è la misura delle dimensioni di un elemento, per fortuna ottenere larghezza e altezza di un elemento è molto facile: basta usare le proprietà offsetWidth e offsetHeight proprie di ogni elemento del DOM:

```
var e = document.getElementById('myEl');
var w = e.offsetWidth;
var h = e.offsetHeight;
```

È comunque da sottolineare che offsetWidth e offsetHeight (così come offsetLeft, offsetTop e offsetParent) non sono proprietà previste dallo standard W3C ma sono state introdotte nel DOM da Microsoft e poi adottate anche dagli altri browser, il loro uso è comunque molto diffuso e comunque visto che è ampiamente supportato.

## 68 Dimensioni di un elemento e visibilità

Una notevole fonte di errori e grattacapi è data dall'utilizzo di offsetWidth e offsetHeight su elementi che sono stati nascosti attraverso la proprietà CSS display="none". Come abbiamo detto nel punto 63 display rende invisibile l'elemento a cui si applica riempiendo lo spazio che occupava con gli altri elementi, ciò comporta che quando si richiama offsetWidth e offsetHeight di un elemento con display="none" il valore sarà sempre 0. Si consideri quindi il seguente codice:

```
var oDiv = document.createElement('div');
oDiv.innerHTML= "nuovo DIV";
oDiv.style.width="100px";
oDiv.style.display = "none";
```

**Figura 66-4:** Offset corretto

```
document.body.appendChild(oDiv);
alert(oDiv.offsetWidth);
```

ebbene il messaggio in questo caso non darà il valore 100 che abbiamo impostato, ma 0. Volendo calcolare una proprietà di misura di un elemento mantenendolo nascosto bisogna quindi nasconderlo prima con visibility, calcolare la misura e quindi resettare visibility e reimpostare a "none" display:

```
var oDiv = document.createElement('div');
oDiv.innerHTML= "nuovo DIV";
oDiv.style.width="100px";
oDiv.style.visibility = "hidden";
document.body.appendChild(oDiv);
var offWidth = oDiv.offsetWidth;
oDiv.style.display = "none";
oDiv.style.visibility = "visible";
alert(offWidth);
```

## 69 Offset nei vari browser

Purtroppo la misura degli elementi DOM è tutt'altro che una scienza esatta! Ogni browser interpreta queste proprietà un po' a modo suo, si consideri ad esempio questo elemento:

```
<div style="width:100px;margin:2px;border:1px solid">un DIV</div>
```

abbiamo quindi:

- una larghezza di 100px
- un bordo di 1px per ogni lato
- un margine di 2px per ogni lato

ebbene per Firefox e Opera offsetWidth è:

```
larghezza + (bordo*2) + margine(*2) = 106
```

mentre per IE solo:

```
larghezza=100
```

In altre parole IE non considera i bordi e i margini mentre altri browser sì. Per evitare questi inconvenienti spesso si ricorre a funzioni che cercano di trovare una misura uniforme per tutti, come :

```
var isIE = (navigator.userAgent.indexOf('MSIE')!=-1);
function realOffsetWidth (e){
    if(!isIE) return e.offsetWidth;
    var ml = parseInt( e.style.marginLeft || 0);
    var mr = parseInt( e.style.marginRight || 0);
    var bl = parseInt( e.style.borderLeftWidth || 0);
    var br = parseInt( e.style.borderRightWidth || 0);
    return e.offsetWidth + ml + mr + bl + br;
}
function realOffsetHeight (e){
    if(!isIE) return e.offsetHeight;
    var mt = parseInt( e.style.marginTop || 0);
    var mb = parseInt( e.style.marginBottom || 0);
```

```
        var bt = parseInt( e.style.borderTopWidth || 0);
        var bb = parseInt( e.style.borderBottomWidth || 0);
        return e.offsetHeight + mt + mb + bt + bb;
    }
```

Comunque, anche così, la sicurezza al 100% non c'è (d'altra parte siamo in un campo dove non esistono standard) l'unica cosa è testare il proprio codice con diversi browser e apportare i correttivi necessari.

# 3.4 TABLE OBJECT MODEL

## 70 Gestire le tabelle

Come strumento di manipolazione dell'HTML il DOM dispone di un sottoinsieme di strumenti per la gestione delle tabelle chiamato Table Object Model. Ovviamente anche sugli elementi TABLE, TR, TD ecc... rimangono validi tutti i metodi generali del DOM (createElement, appendChild, removeChild ecc...) ma troviamo anche proprietà e funzioni specifiche,se ad esempio dovessimo creare una tabella con il DOM generale scriveremmo:

```
var oTable = document.createElement("table"); //<table>
var oTr = document.createElement("tr"); //<tr>
var oTd = document.createElement("td"); //<td>
oTr.appendChild(oTd); //</td>
oTable.appendChild(oTr); //</tr>
document.body.appendChild(oTable); //</table>
con il Table Object Model invece:
var oTable = document.createElement("table"); //<table>
var oTr = oTable.insertRow(-1);//<tr>
var oTd = oTr.insertCell(-1);//<td>
document.body.appendChild(oTable); //</table>
```

La creazione di TABLE resta uguale a quella consueta, però TABLE (ta-

bella) e TR (riga) hanno in più delle collection specializzate (rows e cells) che contengono i sottoinsiemi righe e celle e rispettivi metodi (insertRow e insertCell) per aggiungere nuovi elementi.

## 71 Inserire una riga in una tabella

Come abbiamo visto l'inserimento di una nuova riga nella tabella con Table Object Model si effettua con

```
var oTr = oTable.insertRow(-1);
```

L'argomento numerico di insertRow indica in quale posizione dell'insieme rows effettuare l'inserimento, -1 indica che deve essere effettuato alla fine dell'insieme. Il parametro numerico è opzionale in IE ma obbligatorio in Firefox quindi consigliamo di usarlo sempre.

## 72 Cancellare una riga da una tabella

La cancellazione di una riga da una tabella avviene con deleteRow proprio dell'elemento TABLE, la funzione richiede come argomento l'indice della riga da cancellare:

```
oTable.deleteRow(0); //cancella prima riga
```

## 73 Inserire una cella in una riga

Analogamente a quanto abbiamo visto per la riga anche l'inserimento della cella (elemento TD) avviene richiamando il metodo insertCell sulla riga che deve contenere la nuova cella, anche qui il parametro sta indicare la posizione della cella (-1 per accodarla):

```
var oTd = oTr.insertCell(-1);
```

La cosa strana è che Table Object Model non dispone di un metodo specifico per inserire una cella di tipo TH (l'header della tabella), in questo caso occorre ricorrere ai normali metodi del DOM:

```
var oTh = document.createElement("th");
oTr.appendChild(oTh);
```

## 74 Cancellare una cella da una riga

Questa operazione può essere effettuata attraverso il metodo deleteCell di una riga (elemento TR), la funzione richiede come argomento l'indice della cella da cancellare:

```
oTr.deleteCell(0);//cancella la prima cella
```

# 3.5 EVENTI

## 75 Aggiungere un gestore di evento

In conseguenza delle varie azioni che l'utente compie sulla pagina il browser invia a Javascript dei messaggi di notifica, se nel codice abbiamo impostato una funzione associata a quel determinato evento questa sarà eseguita. Quasi tutti gli elementi del DOM hanno associati degli eventi, i più comunemente utilizzati sono: onload e onunload per l'oggetto WINDOW; onclick, onmouseover, onmouseout per gli elementi HTML in generale; onchange per le SELECT, onsubmit per le FORM; onkeyup, onkeydown per gli INPUT. La funzione associata al verificarsi dell'evento è detta funzione di callback (vedi punto 17) o gestore (in inglese handler). Le tecniche per associare l'evento ad un gestore sono tre:

- associazione dichiarativa, nell'HTML
- associazione singola per assegnazione
- associazione multipla con i listener

## 3.5.1 Associazione dichiarativa

È la più semplice e consiste nell'inserire il codice direttamente come attributo nell'elemento HTML, vediamo ad esempio come assegnare all'evento onclick di un bottone del codice che mostra un messaggio:

```
<button onclick="alert('Hello World')">Clicca qui</button>
```

È anche possibile scrivere nell'attributo del gestore di evento codice anche in più righe:

```
<button onclick="
    var dt =new Date();
    alert('Oggi è il ' + dt.toLocaleString());
">Clicca qui</button>
```

Anche se questa sintassi è da evitare, in quanto rende di difficile lettura sia il codice Javascript che quello HTML. Una particolarità è quella dell'uso di virgolette nel codice di questo tipo, abbiamo visto che come delimitatore di stringa abbiamo utilizzato, all'interno del codice Javascript, la virgoletta singola, ciò perché il codice seguente avrebbe provocato un errore:

```
<button onclick="alert("Hello World")">Clicca qui</button>
```

Poiché infatti come delimitatore di attributi nell'HTML sono state qui usate le virgolette doppie il codice che abbiamo scritto verrebbe interpretato:

```
<button onclick="alert(">Clicca qui</button>
```

ovvero come alert( che in Javascript sarebbe errato sintatticamente. Quindi per le stringhe dentro il codice Javascript dell'associazione di eventi dichiarativa occorre sempre usare il tipo di virgolette non usate per delimitare gli attributi nell'HTML. Quindi sono valide sia :

```
<button onclick="alert('Hello World')">Clicca qui</button>
```

che

```
<button onclick='alert("Hello World")'>Clicca qui</button>
```

Ma ci sono casi in cui la stringa stessa contiene le virgolette, come quello ad esempio in cui il messaggio che vogliamo mostrare sia proprio "Hello World" e non Hello World, per i motivi sopra accennati questa dichiarazione non funzionerebbe:

```
<button onclick="alert('"Hello World"')">Clicca qui</button>
```

ma non funzionerebbe nemmeno usare il carattere di escape “\ ” :

```
<button onclick="alert('\"Hello World\"')">Clicca qui</button>
```

perché l'errore di sintassi è sull' HTML non su Javascript. Il problema si risolve allora sostituendo il carattere delle doppie virgolette con la corrispondente entità HTML ovvero " :

```
<button onclick="alert('&quot;Hello World&quot;')">Clicca qui</button>
```

Un ultima cosa : nel contesto del codice della dichiarazione this è riferito all'elemento HTML (corrispondente al relativo oggetto DOM) nel quale è dichiarato. Nel nostro caso quindi this sarà il bottone stesso e quindi scrivendo:

```
<button onclick="alert(this.value)">Clicca qui</button>
```

il nostro messaggio sarà la proprietà value di Button, in questo caso “Clicca qui”. Ovviamente potremo usare this anche per impostare delle proprietà dell'elemento come il colore del testo:

```
<button onclick="this.style.color='red'">Clicca qui</button>
```

## 3.5.2 Associazione singola per assegnazione

L'associazione dichiarativa, anche se resta un metodo molto usato,

non si presta però in quei casi in cui l'elemento viene creato proprio attraverso i metodi del DOM. Se ad esempio creiamo dinamicamente un elemento DIV:

```
var oDiv = document.createElement('div');
var text = 'Clicca qui';
oDiv.appendChild(document.createTextNode(text));
document.body.appendChild(oDiv);
```

per assegnare l'evento occorre associare una funzione esistente o creata per l'occasione, quindi :

```
function sayHello (){
    alert('Hello World');
}
oDiv.onclick = sayHello;
```

oppure

```
oDiv.onclick = function (){
    alert('Hello World');
}
```

L'associazione singola, oltre che per gli elementi creati dinamicamente, può essere effettuata anche per elementi esistenti nell'HTML referenziati per il loro ID (vedi punto 45), quindi:

```
<div id="div1">Clicca qui</div>
<script>
document.getElementById('div1').onclick = function (){
    alert('Hello World');
}
</script>
```

## 3.5.3 Associazione multipla con i listener

L'associazione singola è già un passo avanti rispetto alla sintassi dichiarativa perché consente di separare il codice HTML da quello Javascript permettendo di creare codice riadattabile a più situazioni diverse (le c.d. librerie). Il suo limite però sta nel rapporto 1:1 con l'elemento, cioè con l'associazione singola possiamo assegnare all'evento una sola funzione alla volta. Pensiamo al caso di due script esterni collegati alla pagina, script1 e script2, che assegnano tutti e due una funzione di callback per l'evento window.onload:

```
//codice script1
window.onload = function(){
alert('ciao dal primo script');
}
//codice script2
window.onload = function(){
alert('ciao dal secondo script');
}
```

nella pagina web i due script saranno inclusi con:

```
…
<head>
    <script src="script1.js" type="text/javascript"></script>
    <script src="script2.js" type="text/javascript"></script>
</head>
…
```

ebbene, in esecuzione il secondo script sovrascrive il gestore definito nel primo script, il risultato sarà che verrà mostrato solo il secondo messaggio. Da qui nasce l'esigenza di poter associare più gestori allo stesso evento. Purtroppo però, come in molti altri casi, le modalità per realizzare l'associazione multipla sono diverse tra Internet

Explorer e i browser che seguono più strettamente gli standard W3C come Firefox e Opera. In Internet Explorer l'associazione multipla di realizza attraverso il metodo attachEvent proprio di ogni oggetto del DOM, la sintassi è:

```
object.attachEvent(nomeEvento, funzioneCallback)
```

dove :

- nomeEvento è il nome dell'evento (onclick,onload ecc…)
- funzioneCallback è il riferimento alla funzione o la funzione creata direttamente

per cui potremmo avere sia:

```
function callback(){
alert('ciao dal secondo script');
}
window.attachEvent('onclick',callback);
```

che

```
window.attachEvent('onclick', function(){
alert('ciao dal secondo script');
});
```

I browser che seguono più fedelmente le specifiche degli standard W3C realizzano invece l'associazione multipla attraverso il metodo addEventListener (Opera per la verità implementa entrambi i sistemi) con sintassi:

```
object.addEventListener (nomeEvento, funzioneCallback,useCapture)
```

dove:

- nomeEvento è il nome dell'evento senza il prefisso "on" (click, load ecc…)
- funzioneCallback è il riferimento alla funzione o la funzione creata direttamente
- useCapture è un valore booleano che indica se l'evento può bloccare o meno gli eventi dello stesso tipo associati a elementi nidificati

la sintassi quindi sarà:

```
function callback(){
alert('ciao dal secondo script');
}
window.addEventListener ('onclick',callback,false);
```

oppure

```
window.addEventListener ('onclick', function(){
alert('ciao dal secondo script');
},false);
```

Il terzo argomento del metodo addEventListener (useCapture) richiede qualche spiegazione in più. Nello standard W3C ad ogni callback viene assegnato, in fase di chiamata, automaticamente l'oggetto Event che può essere utilizzato dalla funzione di callback stessa. Quindi, se useCapture è definito true, la funzione di callback potrà utilizzare il metodo stopPropagation dell'oggetto Event per impedire che vengano generati gli eventi dello stesso tipo sottostanti. Ad esempio, se abbiamo un frammento HTML di questo tipo:

```
<div id="div2" style="border:1px solid">
    <div>uno</div>
    <div onclick="alert('Hello World')">due</div>
</div>
```

a cui associamo questo gestore degli eventi:

```
var oDiv = document.getElementById('div2');
var callback = function  (e){
      alert('secondo messaggio');
}
oDiv.addEventListener('click',callback,false);
```

cliccando sul secondo DIV interno appariranno sia il messaggio definito nella funzione di callback che quello definito con sintassi dichiarativa. Per impedire che ciò accada dovremmo usare:

```
var oDiv = document.getElementById('div2');
var callback = function  (e){
      alert('secondo messaggio');
      e.stopPropagation();
}
oDiv.addEventListener('click',callback,true);
```

Per realizzare l'associazione multipla  in modo che funzioni sia con Internet Explorer che con gli altri browser si è quindi costretti al solito uso di if:

```
if ('addEventListener' in oDiv) {
      oDiv.addEventListener('click',callback,false);
}
else if('attachEvent' in oDiv) {
      oDiv.attachEvent('onclick',callback);
}
```

La tecnica in sé non è difficile, però rende il codice piuttosto "sporco", una soluzione potrebbe essere quella di introdurre la funzione attachEvent come wrapper di addEventListener per i browser confor-

mi al W3C che supportano l'accesso ai prototipi del DOM:

```
if('attachEvent' in window) {
Window.prototype.attachEvent =
Document.prototype.attachEvent =
HTMLElement.prototype.attachEvent = function(name, handler){
this.addEventListener(name.slice(2), handler, false);
//nota: al nome viene tolto il prefisso 'on' come richiede //addEventListener
}
}
```

In questo modo la funzione attachEvent viene aggiunta agli oggetti window, document ed a tutti gli elementi HTML, e sarà quindi possibile utilizzare attachEvent anche su Firefox.

## 76 Gestire l'evento onsubmit delle Form

L'evento onsubmit delle Form è molto importante perché consente di verificare i dati immessi dall'utente prima che essi vengano trasmessi al server. Anche se resta buona norma verificare la correttezza dei dati anche lato server, è comunque preferibile avvertire l'utente di qualche problema che si potrebbe verificare e dargli modo di correggerlo senza dover fare il post. Supponiamo quindi di avere la seguente Form:

```
<form id="myForm" action="/serverProcess" method="post">
    <input type="text" name="testo"/>
    <input type="submit" value="invia"/>
</form>
l'azione definita per l'evento onsubmit potrebbe essere quindi:
var oForm = document.getElementById('myForm');
oForm.onsubmit = function (){
    if(oForm.testo.value==''){
    alert("testo vuoto");
    return false;
```

```
    }
    return true;
}
```

se il controllo verifica qualcosa che non va, la funzione restituisce false ed l'invio si blocca. La gestione dell'evento onsubmit si presta a diversi errori usando la sintassi dichiarativa. Mettiamo ad esempio di aver definito una funzione di controllo separata:

```
function checkForm(){
    var oForm = document.getElementById('myForm');
    if(oForm.testo.value==''){
    alert("testo null");
    return false;
    }
    return true
}
```

Usando la sintassi dichiarativa si è portati ad associarla in questo modo:

```
<form id="myForm" action="/serverProcess" method="post"
                                        onsubmit="checkForm()">
```

così però, in caso di controllo negativo, il messaggio verrebbe sì mostrato, ma il post sarebbe comunque eseguito. Per capire perché questo avviene bisogna pensare che l'associazione dichiarativa è equivalente alla costruzione della funzione effettuata con l'operatore new e quindi sarebbe equivalente a :

```
oForm.onsubmit=new Function("checkForm()");
```

Ora, questo va bene per la grande maggioranza dei gestori di eventi, ma non per onsubmit, perché questo, per bloccare l'invio dei da-

ti, richiede il valore di ritorno false. Perché il codice venga eseguito correttamente dovremmo scrivere allora:

```
<form id="myForm" ... onsubmit="return checkForm()">
```

## 77 Il classico rollOver

Gli eventi onmouseover e onmouseout danno la possibilità di implementare una delle più classiche applicazioni di Javascript, quella del rollover su bottoni. Lo possiamo fare nella forma più banale con l'associazione dichiarativa:

```
<button onmouseover="this.style.color='red'"
                    onmouseout="this.style.color='black'">
Clicca qui!
</button>
```

in questo modo impostiamo il testo di colore rosso al passaggio del mouse. Già un piccolo passo avanti è quello di associare delle regole CSS anziché impostare le singole proprietà:

```
<style>
    .normalButton {color : black}
    .overButton {color : red}
</style>
...
<button class="normalButton" onmouseover="this.className
        ='overButton'" onmouseout="this.className='normalButton'">
Clicca qui!
</button>
```

in questo modo abbiamo il vantaggio di poter impostare agevolmente più attributi di formattazione per volta, semplicemente definendoli nelle classi CSS. Resta tuttavia il grosso inconveniente di

dover ripetere la dichiarazione per ogni BUTTON che vogliamo dotare di questa funzionalità. Si potrebbe agire quindi su un gruppo di controlli applicando gli eventi con un ciclo for:

```
<script language="javascript" type="text/javascript">
function applyRollover (){
    var ids = ['btn1','btn2','btn3'];
    for(var i=0;i<ids.length;i++){
        var btn = document.getElementById(ids[i]);
        btn.className ='normalButton';
        btn.onmouseover = function  (){
            this.className='overButton';
        }
        btn.onmouseout = function  (){
            this.className='normalButton';
        }
    }
}
window.onload = function () {
    applyRollover();
}
</script>
<button id="btn1">Clicca sul bottone 1</button>
<button id="btn2">Clicca sul bottone 2</button>
<button id="btn3">Clicca sul bottone 3</button>
```

Possiamo tuttavia fare ancora uno sforzo in più generalizzando l'operazione in una funzione riapplicabile:

```
function applyRollover (classNormal,classOver){
    //gli id degli elementi vanno aggiunti ai parametri
    for(var i=2;i<arguments.length;i++){
        var element = document.getElementById(arguments[i]);
```

```
            element.className =classNormal;
            element.onmouseover = function  (){
                  this.className=classOver;
            }
            element.onmouseout = function  (){
                  this.className=classNormal;
            }
      }
}
window.onload = function () {
      applyRollover('normalButton','overButton','btn1','btn2','btn3');

}
```

in questo modo chiamando il metodo applyRollover basterà indicare le sue classi CSS e gli id degli elementi a cui applicare il rollover.

# 3.6 LOCALIZZAZIONE

## 78 Conoscere la lingua impostata dall'utente

Le informazioni sulla lingua del browser dell'utente sono fornite dall'oggetto navigator, come spesso avviene però, ci sono delle differenze tra i vari browser : IE fornisce la proprietà browserLanguage, Firefox invece la chiama language, Opera invece le ha entrambe.
Il modo per conoscere sicuramente l'informazione sulla lingua è:

```
var l = ('language' in navigator)?navigator.language:
                                            navigator.browserLanguage;
```

il valore restituito è la sigla della lingua (it per italiano, en per inglese ecc…)  secondo la codifica internazionale.

# 79 Localizzare l'interfaccia utente - libreria

Uno dei problemi più grossi nello sviluppo di librerie e controlli in Javascript è quello di trovare un sistema che permetta che l'interfaccia utente sia facilmente adattabile ad altre lingue senza duplicare il codice. Poniamo di dover scrivere un controllo HTML da inserire in una libreria generica che potrà essere utilizzata da utenti di diversi paesi. Come esempio prendiamo una semplice interfaccia HTML, ovvero una Form di richiesta dati, come questa:

```
<form action="#" method="post" id="frmDati">
<table cellspacing="0" cellpadding="0"
                                  align="center" class="formTable">
   <tr class="container">
      <td title="Immetti il valore Nome" class="legend">Nome</td>
      <td class="input"><input type="text"
                           value="" name="name" id="name"/></td>
   </tr>
   <tr class="container">
      <td title="Immetti il valore
                           Cognome" class="legend">Cognome</td>
      <td class="input"><input type="text" value=""
                           name="cognome" id="cognome"/></td>
   </tr>
   <tr class="container">
   <td title="Immetti il valore Professione"
                              class="legend">Professione</td>
   <td class="input">
   <select id="professione">
   <option value="1">Libero professionista</option>
   <option value="2">Dipendente</option>
   <option value="3">Pensionato</option>
   <option value="4">Altro</option>
```

```
    </select>
    </td>
    </tr>
    <tr class="container">
        <td title="Immetti il valore Città" class="legend">Città</td>
        <td class="input"><input type="text" value="" name="citta"
                                                    id="citta"/></td>
    </tr>
    <tr class="container">
        <td title="Immetti il valore Provincia"
                                        class="legend">Provincia</td>
        <td class="input"><input type="text" value="" name="prov"
                                                    id="prov"/></td>
    </tr>
    <tr>
        <td colspan="2" class="inputSend">
        <input type="submit" value="Invia dati" name="" id=""/>
        </td>
    </tr>
</table>
</form>
```

Notiamo che il codice è "infarcito" di stringhe in lingua italiana, sia impostate come attributi che come testo. Se dovessimo distribuire il nostro controllo ad un pubblico internazionale dovremmo quindi realizzarne tante copie quante sono le lingue supportate, se poi dovessimo cambiare qualcosa nella struttura dovremmo anche apportare le stesse modifiche nelle copie ...decisamente non è questa la strada! Quella che dovremmo ottenere è la possibilità di scrivere dei "segnaposti" al posto delle stringhe che poi vengano sostituiti in fase di esecuzione, e cioè:

```
<form id="frmDati" method="post" action="#">
```

```
<table class="formTable" align="center" cellpadding="0"
                                                 cellspacing="0">
    <tr class="container">
        <td class="legend" title="$legendTitle $nome">$nome</td>
        <td class="input"><input type="text" id="name"
                                    name="name" value=""></td>
    </tr>
    <tr class="container">
        <td class="legend" title="$legendTitle $cognome">
                                                  $cognome</td>
        <td class="input"><input type="text" id="cognome"
                                 name="cognome" value=""></td>
    </tr>
    <tr class="container">
    <td class="legend" title="$legendTitle $professione">
                                              $professione</td>
    <td class="input">
    <select id="professione">
    <option value="1">$professione_prof</option>
    <option value="2">$professione_dip</option>
    <option value="3">$professione_pens</option>
    <option value="4">$professione_altro</option>
    </select>
    </td>
    </tr>
    <tr class="container">
        <td class="legend" title="$legendTitle $citta">$citta</td>
        <td class="input"><input type="text" id="citta" name=
                                          "citta" value=""></td>
    </tr>
    <tr class="container">
        <td class="legend" title="$legendTitle $prov">$prov</td>
        <td class="input"><input type="text" id="prov" name=
```

```
                                                  "prov" value=""></td>
    </tr>
    <tr>
        <td class="inputSend" colspan="2">
        <input type="submit" id="" name="" value="$invia">
        </td>
    </tr>
</table>
</form>
```

Usando un linguaggio lato server come ASP o PHP ovviamente questo non sarebbe un problema (basterebbe sostituire ai segnaposti dei tag di scrittura di variabili lato server), ma in questo caso vincoleremmo gli utilizzatori del nostro controllo ad usare quel determinato linguaggio restringendo così la platea dei nostri "estimatori". La cosa però può essere fatta anche in Javascript utilizzando il DOM.
Per prima cosa, nel codice Javascript, dichiariamo l'oggetto CurrentLang che sarà un po' la nostra "banca dati" delle stringhe di testo:

```
var CurrentLang = {
    addDefinition: function (name,value){
        this[name]=value;
    },
    addLangPack:function (langPack){
        for(m in langPack){
            this.addDefinition(m,langPack[m]);
        }
    },
//esempio definizioni:
    pageTitle:'Esempio localizzazione',
    legendTitle:'Immetti il valore',
    nome:'Nome'
}
```

CurrentLang dispone di due metodi fondamentali:

- addDefinition che consente di aggiungere una nuova proprietà all'oggetto stesso
- addLangPack che consente di prendere un oggetto e copiarne tutte le proprietà dentro l'oggetto stesso

Come esempio abbiamo poi inserito anche alcune definizioni che saranno, come vedremo, i valori di default. Successivamente occorre impostare una funzione dal nome breve e facilmente memorizzabile, $lang , che ci consenta di recuperare i dati da CurrentLang, consentendoci di impostare anche un valore di default in caso in cui la definizione non venga trovata:

```
function $lang (name,vDefault){
      var vDefault = (vDefault==null)?name:vDefault;
      if(name in CurrentLang)         return CurrentLang[name];
      else return vDefault;
}
```

A questo punto non resta che impostare l'oggetto, che chiameremo Localizer, che farà tutto il lavoro di trasformazione dei segnaposti impostati con le definizioni effettive:

```
var Localizer = {
//carica uno script esterno inserendolo nel documento corrente
      loadExternal : function  (packageName){
            document.write('<scr' + 'ipt src=' + packageName +
             ' language="javascript" type="text/javascript"></scr' + 'ipt>');
      },
//fornisce un metodo (eventualmente sostituibile) per reperire il nome
//dello script esterno partendo dal codice del linguaggio
      getPackageName : function (languageId){
```

```
        return "lang_" + languageId + ".js";
    },
//trova il codice del linguaggio attualmente in uso nel browser
    browserLanguage : ('language' in navigator)?navigator.language:
                                        navigator.browserLanguage,
//inizializza l'oggetto caricando un determinato script esterno
    init:function  (languageId){
        var languageId = languageId || this.browserLanguage;
        this.loadExternal(this.getPackageName(languageId));
    },
//il pattern di sostituzione dei segnaposto che saranno nella forma $name
    replacePattern :  /\${1}[\w_]+/g,
//la funzione di replace usata dalla Regular Expression
    replaceFunction: function (s){
        var name = s.substr(1);
        return $lang(name,s);
    },
//funzione che avvia il replace a partire da un dato nodo HTML
    _runReplace : function  (startNode){
        //crea riferimenti interni a replacePattern e replaceFunction in
        //modo che siano disponibili anche alle funzioni interne
        var replacePattern = this.replacePattern;
        var replaceFunction = this.replaceFunction;
        //analizza document.title che non è modificabile attraverso
        //i normali metodi DOM
        if(document.title.indexOf('$')!=-1){
            document.title =
                document.title.replace(replacePattern,replaceFunction);
        }
        // isAllowedNode: funzione interna che serve ad escludere
// dal replace alcuni elementi della pagina
        // come <script>, <style> ecc…
        var isAllowedNode = function  (nd){
```

```
            var nodeName = nd.nodeName.toLowerCase();
            for(var i=1;i<arguments.length;i++){
                if(arguments[i].toString().toLowerCase()==nodeName) {
                    return false
                }
            }
            return true;
        }
        // walkNodes: funzione interna che effettua il replace in un nodo
        var walkNodes = function (nd){
            if(!isAllowedNode(nd,'script','style','link')) return;
            // replaceValue : funzione interna che applica la
// trasformazione ai nodi di testo e agli attributi
            var replaceValue = function  (currentNode){
                try{
                    if( currentNode==null ||
                        currentNode.nodeValue==null ||
                        typeof(currentNode.nodeValue)!= 'string'||
                        currentNode.nodeValue.indexOf('$')==-1)
                                                        return ;
                    var s = currentNode.nodeValue.toString().replace
                                (replacePattern,replaceFunction);
                    if(currentNode.nodeType == 3) { //textNode
                    /*
```

l'assegnazione di nodeValue per textNode non è ammessa, funziona in Firefox ma dà errore in IE quindi occorre creare un nuovo textNode appenderlo e cancellare il vecchio

```
                    */
                        var tn = document.createTextNode(s);
                        var pn = currentNode.parentNode;
                        pn.removeChild(currentNode);
```

```
                    pn.appendChild(tn);
                }
                else if(currentNode.nodeType == 2) { //attribute
                    currentNode.nodeValue = s;
                }
            }
            catch(e){/*evita errori imprevisti*/}
        } //fine replaceValue
        //controlla il tipo di nodo corrente
        switch(nd.nodeType) {
            case 3: /*text node*/
                replaceValue(nd);
            break;
            case 1: /*element*/
            //in un elemento vengono sostituiti i valori degli
            //attributi impostati
                for(var i=0;i<nd.attributes.length;i++) {
                    var a = nd.attributes[i];
                    if(a.specified || a.nodeName=='value'){
                        replaceValue(a);
                    }
                }
            break;
        }
        /*
```

A questo punto la funzione richiama se stessa per tutti i nodi figli di quello corrente

```
*/
        var childs = nd.childNodes;
        for(var i=0;i<childs.length;i++){
            var childNode = childs[i];
```

```
                    walkNodes(childNode);
                }
            } // fine walkNodes
            walkNodes(startNode); //esegue walkNodes
    },
//funzione "pubblica" che esegue il replace a partire da un dato nodo
//se startNode non è impostato parte da <body>
    replace : function (startNode){
            var startNode = startNode || document.body;
            this._runReplace(startNode);
    }
};
```

## 80 Localizzare l'interfaccia utente - uso

Per usare la libreria per la localizzazione occorre:

- inizializzarla usando la funzione init con il codice lingua appropriato, questa caricherà il file esterno contenente le definizioni che verranno aggiunte a CurrentLang
- richiamare il metodo replace per avviare la sostituzione dei segnaposto con le corrispondenti definizioni contenute in CurrentLang

Prepareremo quindi, prima di tutto, un file esterno per la lingua italiana (codice lingua "it") che, secondo la nostra convenzione sarà chiamato lang_it.js, in esso sarà richiamato il metodo addLangPack di CurrentLang:

```
/*IT Lang Pack*/
CurrentLang.addLangPack(
    {
    pageTitle:'Esempio localizzazione',
    legendTitle:'Immetti il valore',
    nome:'Nome',
    cognome:'Cognome',
```

```
    professione:'Professione',
    professione_prof:'Libero professionista',
    professione_dip:'Dipendente',
    professione_pens:'Pensionato',
    professione_altro:'Altro',
    citta:'Città',
    prov:'Provincia',
    invia:'Invia dati'
    });
```

per altre lingue analogamente prepareremo altri file (lang_en.js ecc…):

```
/*EN Lang Pack*/
CurrentLang.addLangPack(
    {
    pageTitle:'localization example',
    legendTitle:'Type the value',
    nome:'Name',
    cognome:'Last name',
    professione:'Job'
    //ecc…
    });
```

Torniamo quindi al nostro documento principale e, nel codice Javascript, richiamiamo il metodo init di Localizer con il codice lingua appropriata:

```
Localizer.init('it');
//oppure
//Localizer.init('en'); ecc…
```

Quindi, nel gestore dell'evento onload di window (per essere sicuri che tutto il DOM sia già stato caricato), richiamiamo replace (senza argomenti in modo da farlo partire da <BODY>):

```
window.onload = function () {
    Localizer.replace();
}
```

Per gli oggetti che vengono creati dinamicamente attraverso il DOM possiamo usare direttamente la funzione globale $lang :

```
var oDiv = document.createElement('DIV');
oDiv.innerHTML= $lang('legendTitle');
document.body.appendChild(oDiv);
```

oppure, nei casi in cui viene appeso un frammento HTML, richiamare replace sul nodo:

```
var oDiv = document.createElement('DIV');
oDiv.innerHTML= '<span>$legendTitle</span>';
document.body.appendChild(oDiv);
Localizer.replace(oDiv);
```

# 3.7 SORTING DI TABELLE

## 81 Rendere ordinabili le tabelle - libreria

Una problematica piuttosto comune è quella del sorting delle tabelle. Proprio per la generalità dei casi in cui c'è esigenza di applicare questa tecnica andremo a scrivere una libreria riapplicabile ad una pluralità di situazioni. Andremo perciò a creare un oggetto che ha il compito di aggiungere ad una tabella le funzionalità di sorting.
L'oggetto, che chiameremo TableSorter, ha al suo interno:

- **currentConfig** - l'oggetto di configurazione che riporta varie proprietà ed un metodo addConfig che consente di espanderlo con configurazioni aggiunte dall'utente (numero di righe fisse da non ordinare, stili ecc…)

- **compare** - funzione che ha il compito di comparare due valori per il sorting restituendo il valore numerico appropriato
- **parseDate** - funzione che ha il compito di analizzare una stringa con le Regular Expressions per trasformarla in Date.
- **apply** - funzione principale che ha il compito di estendere una qualsiasi tabella aggiungendo i metodi per il sorting.

Lo schema dell'oggetto (con apply vuoto) si presenta quindi:

```
var TableSorter = {
    currentConfig : {
        addConfig:function (configObject){
            for(prop in configObject){
                this[prop]=configObject[prop];
            }
        },
        columnTypes:[],
        headerRows:1, //numero righe fisse headers
        footerRows:0,  //numero righe fisse footer
        unsortedClassName:'',
        sortedAscClassName:'',
        sortedDescClassName:''
    },
    compare : function  (v1,v2,reverse){
        var r=0;
        if(v1<v2) r = -1;
        else if(v1>v2) r = 1;
        else  r = 0;
        return r * reverse;
    },
    parseDate : function (re,value,yIndex,mIndex,dIndex){
        if(value==null || !re.test(value)) {
            return new Date(1970,0,1);
```

```
                    }
                else {
                    var arr = re.exec(value);
                    var y = arr[yIndex + 1];
var m = arr[mIndex + 1] -1;
var d = arr[dIndex + 1];
                    return new Date(y,m,d);
                }
        },
        apply : function (oTable,configObject){
        ...
        }
}
```

apply, come abbiamo detto, è il metodo principale di TableSorter. In primo luogo ci preoccuperemo di passare alla tabella la configurazione di default integrata, se presente, da quella personalizzata:
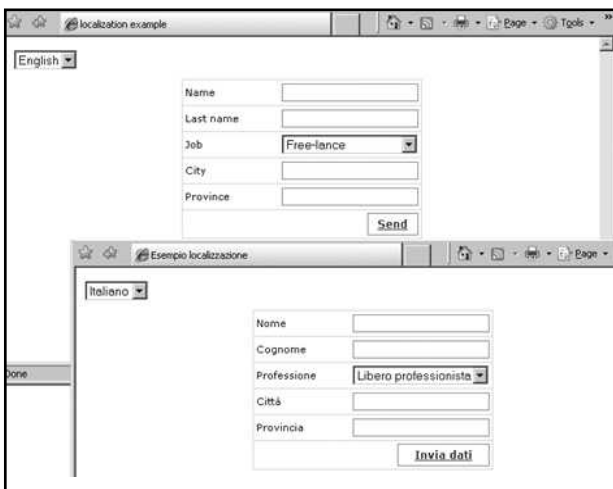
**Figura 80-1:** Il risultato dell'applicazione della nostra libreria

```
        var cfg = oTable.config = this.currentConfig;
        if(configObject!=null) cfg.addConfig(configObject);
```

poi definiamo l'oggetto sortFunctions come proprietà della tabella, sortFunctions conterrà vari tipi di sorting applicabili:

```
        oTable.sortFunctions = {
//testo case insensitive
            text : function  (data,cellIndex,reverse){
                data.sort(function  (item1,item2){
            var v1 = item1.objects[cellIndex].toString().toLowerCase();
            var v2 = item2.objects[cellIndex].toString().toLowerCase();
                    return TableSorter.compare(v1,v2,reverse);
                });
            },
//numerica
            number : function (data,cellIndex,reverse){
                data.sort(function  (item1,item2){
                    var v1 = new Number(item1.objects[cellIndex]);
                    var v2 = new Number(item2.objects[cellIndex]);
                    if(isNaN(v1))v1=0;
                    if(isNaN(v2))v2=0;
                    return TableSorter.compare(v1,v2,reverse);
                });
            },
//data GMA
            dmy : function (data,cellIndex,reverse){
                data.sort(function  (item1,item2){
                    var v1 = item1.objects[cellIndex].toString();
                    var v2 = item2.objects[cellIndex].toString();
                    var re = /(\d{1,2})\D{1}(\d{1,2})\D{1}(\d{4})/;
                    v1 = TableSorter.parseDate(re,v1,2,1,0);
                    v2 = TableSorter.parseDate(re,v2,2,1,0);
```

```
                    return TableSorter.compare(v1,v2,reverse);
               });
          },
//data AMG
          ymd : function (data,cellIndex,reverse){
               data.sort(function  (item1,item2){
                    var v1 = item1.objects[cellIndex].toString();
                    var v2 = item2.objects[cellIndex].toString();
                    var re = /(\d{4})\D{1,}(\d{1,2})\D{1,}(\d{1,2})/;
                    v1 = TableSorter.parseDate(re,v1,0,1,2);
                    v2 = TableSorter.parseDate(re,v2,0,1,2);
                    return TableSorter.compare(v1,v2,reverse);
               });
          },
//Regular Expression
          rex : function (data,cellIndex,reverse,pattern){
               var re=new RegExp(pattern,'img');
               data.sort(function  (item1,item2){
                    var v1 = item1.objects[cellIndex].toString();
                    var v2 = item2.objects[cellIndex].toString();
                    if(re.test(v1)) v1=v1.match(re)[0];
                    if(re.test(v2)) v2=v2.match(re)[0];
                    return TableSorter.compare(v1,v2,reverse);
               });
          }
     }
```

poi si definiranno quali sono il gruppo di righe ordinabili (escludiamo la prima riga ed eventualmente le righe di fondo tabella):

```
     var rows = $A(oTable.rows);//trasforma le rows in Array
     var start =  oTable.start = cfg.headerRows||1;
```

```
        var end = rows.length -  cfg.footerRows;
        var sortableRows = oTable.sortableRows = rows.slice(start,end);
        //le righe sono inferiori al numero di header e footer definiti
        if(rows.length <= (start + cfg.footerRows)) return;

        rows.foreach(function (row){
        //ad ogni riga associamo il riferimento alla tabella
            row.table = oTable;
        });
```

adesso invece prendiamo la prima riga della tabella (quella dei titoli delle colonne) e le associamo le funzionalità di sorting sul click che verrà gestito dalla funzione di callback onsort assegnata sempre all'oggetto oTable successivamente:

```
    var headerRow = oTable.headerRow = rows[0];
    for(var i=0;i<headerRow.cells.length;i++){
        var hdCell = headerRow.cells[i];
        hdCell.row = headerRow;
        hdCell.table = oTable;
        hdCell.className = cfg.unsortedClassName;
        if(hdCell.getAttribute("nosort")!= '') {
            hdCell.attachEvent ('onclick',
                function (){
                    var current =
                        (this==window)?window.event.srcElement:this;
                    var headerRow = current.table.headerRow;
                    var oTable = current.table;
                    for(var i=0;i<headerRow.cells.length;i++){
                    var hdCell = headerRow.cells[i];
                        if(hdCell!=current) {
                    hdCell.className =
                                Table.config.unsortedClassName;
```

```
                        }
                    }
                    oTable.onsort(current);
                });
        }
    }
```

quindi creiamo il nostro repository di dati composto dall'Array data popolato di oggetti item; ogni item conserva il riferimento alla riga della tabella ed ha a sua volta un Array objects composto dai valori letti dalle celle:

```
    var data = oTable.currentData = [];

    sortableRows.foreach(function (row){
        var item = {'row':row};
        item.objects = [];
        $A(row.cells).foreach(function  (cell){
            item.objects[cell.cellIndex] = getTextNode(cell);
        });
        data.push(item);
    });
```

definiamo quindi due proprietà della tabella, currentSortType e currentColumnIndex, che serviranno a memorizzare il tipo di sorting (ascendente o discendente) e la colonna a quale è applicato:

```
        oTable.currentSortType = 'asc';
        oTable.currentColumnIndex = -1;
```

quindi definiamo la funzione di callback onsort che determina quale tipo di sorting applicare, ordina il repository dei dati, sposta di conseguenza le righe della tabella e imposta i CSS della cella contenente il titolo della colonna:

```
oTable.onsort = function (headerCell){
    var cellIndex = headerCell.cellIndex
    //this è nel contesto oTable
    var data = this.currentData;
    if(this.currentColumnIndex == cellIndex) {
        if(this.currentSortType =='asc') this.currentSortType='desc';
        else if(this.currentSortType =='desc') this.currentSortType='asc';
    }
    else this.currentSortType='asc';
    var rev = (this.currentSortType=='asc')?1:-1;
    this.currentColumnIndex = cellIndex;
    var sortType = headerCell.getAttribute("sortType") || 'text';
    if(sortType.indexOf('rex:')!=-1) {
        this.sortFunctions.rex(data,cellIndex,
                                        rev,sortType.replace('rex:',''));
    }
    else if(sortType in this.sortFunctions) {
        this.sortFunctions[sortType](data,cellIndex,rev);
    }
    for(var i=0;i<data.length;i++){
        var sorted = data[i].row;
        sorted.swapNode(this.rows[i + 1]);
    }
    //cambia le classi CSS della cella contenente il titolo della colonna
    headerCell.className = (this.currentSortType=='asc')

      ?this.config.sortedAscClassName: this.config.sortedDescClassName;
}
```

Tutto questo, lo ripetiamo, all'interno della funzione apply di TableSorter. Da sottolineare che il tipo di sorting (su testo, numeri,date, regular expressions ecc...) è determinato semplicemente dagli attributi sortType che mettiamo nelle celle della prima riga della tabella:

```
<table id="myTable">
<tr class="header">
<td sortType="number">progr.</td>
<td>codice</td>
<td sortType="rex:\w+$">nome</td>
<td sortType="ymd">data</td>
</tr>
...
</table>
```

il valore di questi attributi è corrispondente alle funzioni di sorting che abbiamo creato in precedenza nell'oggetto sortFunctions associato alla tabella in apply. L'unica particolarità è nel sorting con le regular expressions che presenta il pattern di estrazione dati preceduto dal prefisso "rex:".

## 82 Rendere ordinabili le tabelle - uso

Al contrario del suo sviluppo l'utilizzo dello script per ordinare le tabelle è molto semplice: otteniamo un riferimento ad una tabella e gli applichiamo il metodo apply di TableSorter aggiungendo come parametro l'eventuale configurazione personalizzata. Ecco un esempio:

```
window.onload = function () {
    var myTable = document.getElementById('myTable');
    TableSorter.apply(myTable,{
        footerRows:1,
        unsortedClassName:'unsorted',
        sortedAscClassName:'sortedAsc',
        sortedDescClassName:'sortedDesc'});
}
```

## 83 Drag and Drop - libreria

Una delle funzionalità più "teatrali" che si possono ottenere combinando il DOM con Javascript è senz'altro il Drag and Drop, ovvero il trascinamen-

to di oggetti con il mouse. La "piccola" libreria che abbiamo sviluppato come esempio consta comunque di più di 300 linee di codice quindi per l'analisi completa rimandiamo al codice allegato o al sito web indicato nelle note finali. Ci limitiamo qui a descrivere i passaggi fondamentali. Nel nostro esempio il Drag and Drop è una funzionalità aggiunta agli elementi del DOM dall'oggetto DragManager. Il nostro obbiettivo è duplice :

- che sia possibile usare una forma dichiarativa nell'HTML come

```
<li dragGroup="list1" dragSource="true">item 1</li>
```

- che sia possibile abilitare il Drag and Drop anche per gli oggetti costruiti dinamicamente con il DOM come

```
var oDiv = document.createElement('div');
oDiv.innerHTML = "Drag me!"
document.body.appendChild(oDiv);
DragManager.registerSource (oDiv);
```

le funzionalità di Drag and Drop sono gestite dall'evento onmousemove dell'oggetto document per il quale, all'interno di DragManager, è stato definito l'handler mousePositionListener:

```
DragManager = {
...
    mousePositionListener:function  (e){
/*
```

il contesto di questa funzione è l'elemento al quale è associato l'evento è per questo che ci si riferisce a DragManager in modo esplicito e non con this

```
*/
/*
```

imposta:

- DragManager.element : elemento sopra il quale si trova il mouse
- DragManager.button : bottone premuto
- DragManager.docX e DragManager.docY : coordinatee del mouse nella pagina

```
*/
        if(e.target){
            //Firefox, Opera e Safari
              DragManager.element = e.target;
              if (DragManager.buttonState){
            /*
```

Il valore del codice che indica quale bottone del mouse è stato premuto è diverso tra IE e gli altri browser, qui lo convertiamo ai valori

**Figura 82-1:** Il sorting applicato a una tabella

di IE perché è utile disporre del codice 0 che indica che non è stato premuto nessun bottone

```
*/
                switch(e.button) {
                    case 0://left button
                        DragManager.button = 1;break;
                    case 1: //middle button
                        DragManager.button = 0;break;
                    case 2://right button
                        DragManager.button = 2;break;
                    default:
                        DragManager.button = 1;
                }
         }
         else {
                DragManager.button = 0;
         }
       if( typeof( e.pageX ) == 'number' ) {
         DragManager.docX = e.pageX;
         DragManager.docY = e.pageY;
       }
       else{
         DragManager.docX = e.clientX;
         DragManager.docY = e.clientY;
       }
     }
     else{
        //IE
       //e = window.event;
         DragManager.element = e.srcElement;
         DragManager.button = e.button;
         if(DragManager.button > 2) {
```

```
                if(DragManager.button==3||DragManager.button=
                    =5||DragManager.button==7) DragManager.button=1;
                else DragManager.button=0;
            }
        DragManager.docX = e.clientX;
        DragManager.docY = e.clientY;
            if( document.documentElement &&
                        ( document.documentElement.scrollTop |
                        | document.documentElement.scrollLeft ) ){
                DragManager.docX += document.document
                                        Element.scrollLeft;
                DragManager.docY += document.document
                                        Element.scrollTop;
            }
            else if( document.body && ( document.body.scrollTop ||
                                document.body.scrollLeft ) ) {
                DragManager.docX += document.body.scrollLeft;
                DragManager.docY += document.body.scrollTop;
            }
        }
        if(DragManager.dragElement==null && DragManager.button==1) {
/*
```

In questo caso l'elemento da spostare non è ancora definito, ma il bottone premuto è quello sinistro, per cui inizia il Drag.

- viene valutato se l'elemento su cui è stato fatto click è impostato come DragSource con DragManager.getDragSource(DragManager.element)
- se l'elemento è un DragSource viene creato un duplicato dell'elemento attraverso createDragElement e assegnato alla variabile dragElement

```
*/
```

```
                //dragStart
                var ds = DragManager.getDragSource
                                        (DragManager.element);
                if(ds)
DragManager.dragElement =DragManager.createDragElement(ds);
        }
        else if(DragManager.dragElement!=null && DragManager.
                                              button!=1) {
/*
```

In questo caso esiste un elemento che si sta spostando e l'utente ha rilasciato il bottone, quindi viene richiamato dragStop che valuterà se l'elemento sopra cui è stato rilasciato è un DropTarget

```
*/
                DragManager.dragStop();
        }
        else if(DragManager.dragElement!=null &&
                                  DragManager.button==1) {
/*
```

Qui siamo in fase di trascinamento quindi viene spostato dragElement (che è il clone dell'elemento originario) e se stiamo passando sopra un target vengono richiamate le eventuali funzioni di callback associate al target

```
*/
                var tg = DragManager.getDragTarget();
                if(tg){
                     if(DragManager.lastTargetOver!=null
&& DragManager.lastTargetOver != tg) {
                          DragManager.lastTargetOver.notifyOut();
                          }
                     DragManager.lastTargetOver = tg;
```

```
                    DragManager.lastTargetOver.notifyOver();
              }
              else {
                    if(DragManager.lastTargetOver!=null)
                                     DragManager.lastTargetOver.notifyOut();
                    DragManager.lastTargetOver = null;
              }
              DragManager.moveDragElement();
         }
      },
...
}
```

I metodi per registrare un elemento come DragSource (che può essere trascinato) o DragTarget (che può ricevere oggetti trascinati) o entrambi (che può essere al tempo stesso DragSource o DragTarget) sono registerSource, registerTarget e registerSourceAndTarget:

```
DragManager = {
...
//registra la sorgente del Drag
      registerSource : function (element){
      if(typeof(element)=='string') element=document.get
                                                  ElementById(element);
            if(element==null) return ;
            element.dragSource=true;
            HtmlUtils.disableSelection(element);
            this.sources.push(element);
      },
//registra il target del Drag

      registerTarget : function (element,callback,dragover,dragout){
      if(typeof(element)=='string') element=document.getElementById
```

```
                                                        (element);
        if(element==null) return ;
        if(typeof(callback)=='function') {
/*
```

Se è stata passata una funzione con l'argomento callback viene associata all'elemento stesso

```
*/
            element.dragCallback = callback;
            }
        else {
            element.dragCallback = function  (){};
            }
        element.dragTarget=true;
/*
```

Viene definita la funzione "withinCoords" che mi dice se l'elemento si trova all'interno delle coordinate X Y del mouse

```
*/
        element.withinCoords = function  (){
            var pos = HtmlUtils.findPos(this);
            var docX = DragManager.docX;
            var docY = DragManager.docY;
            return (
                (docX >= pos.left &&  docX <=
                                    (pos.left + this.offsetWidth)) &&
                (docY >= pos.top && docY <=(pos.top
                                            + this.offsetHeight))
                );
        };
/*
```

Pure qui vengono assegnate le funzioni di dragover e dragout sull'elemento

```
*/
        element.notifyOver = dragover || function  (){};
        element.notifyOut = dragout || function  (){};
        HtmlUtils.disableSelection(element);
        this.targets.push(element);
    },
//registra la sorgente e il target del Drag
    registerSourceAndTarget : function (element,callback,dragover,
                                                            dragout){
        this.registerSource(element);
        this.registerTarget(element,callback,dragover,dragout);
    },
...
}
```

DragManager dispone di due metodi fondamentali per il suo utilizzo, install e startCapture:

- install scorre tutti i nodi della pagina (a partire da quello indicato o da BODY) e valuta gli attributi degli elementi, registrando quelli per cui sono stati impostati gli attributi appropriati.
- startCapture invece avvia i listener sugli eventi.

```
DragManager = {
...
    install:function (fromElement){
        var startElement = fromElement || document.body;
        var elements = startElement.getElementsByTagName('*');
        var getElementFunction = HtmlUtils.getElementFunction;
        var getBoolAttribute = HtmlUtils.getBoolAttribute;
        for(var i=0;i<elements.length;i++){
```

```
                var e = elements[i];
                var isDragSource = getBoolAttribute(e,'dragSource');
                var isDragTarget = getBoolAttribute(e,'dragTarget');
                var isDragBoth = getBoolAttribute(e,'dragBoth');
                var dragOnFx = getElementFunction(e,'dragOn');
                var dragOverFx = getElementFunction(e,'dragOver');
                var dragOutFx = getElementFunction(e,'dragOut');
                e.dragGroup = e.getAttribute("dragGroup");
                if(isDragSource || isDragBoth) this.registerSource(e);
                if(isDragTarget || isDragBoth) {
                        this.registerTarget(e,dragOnFx,dragOverFx,dragOutFx);
                }
        }
    },
    startCapture: function  (){
        if('addEventListener' in document) {
                document.addEventListener("mouseup",
                                        DragManager.mouseUpListener,false);
                document.addEventListener("mousedown",
                                        DragManager.mouseDownListener,false);
                document.addEventListener("mousemove",
                                        DragManager.mousePositionListener,false);
        }
        else {
    document.attachEvent("onmouseup",DragManager.mouse
                                                            UpListener);
    document.attachEvent("onmousedown",DragManager.mouse
                                                            DownListener)
document.attachEvent("onmousemove",DragManager.mousePosition
                                                            Listener);
        }
    },
}
```

## 84 Drag and Drop - uso

L'uso del Drag and Drop è semplice, per prima cosa inseriremo il riferimento alla libreria :

```
<script src="dnd.js" type="text/javascript"></script>
```

Come abbiamo detto la libreria supporta sia la sintassi dichiarativa che la costruzione attraverso il DOM. Con la sintassi dichiarativa imposteremo le proprietà come attributi dei target:

```
<ul>
    <li dragGroup="list1" dragBoth="true" dragOn="dragOn"
                dragOver="dragOver" dragOut="dragOut">item 1</li>
    <li dragGroup="list1" dragBoth="true" dragOn ="dragOn"
                dragOver="dragOver" dragOut="dragOut">item 2</li>
</ul>
```

Gli attributi impostabili sono :

- **dragSource** - (true o false) l'elemento è trascinabile o meno
- **dragTarget** - (true o false) l'elemento può ricevere elementi trascinati o meno
- **dragBoth** - (true o false) l'elemento è sia dragSource che dragTarget
- **dragGroup** - (nome) l'elemento appartiene al gruppo (può essere utilizzato nella funzione di callback per determinare se DragSource e DragTarget appartengono allo stesso gruppo)
- **dragOn** - (nome di funzione) funzione di callback per gestire l'azione da compiere quando un DragTarget riceve un DragSource
- **dragOver** - (nome di funzione) funzione di callback per gestire l'azione da compiere quando un DragSource passa sopra l'elemento DragTarget
- **dragOut** - (nome di funzione) funzione di callback per gestire l'a-

zione da compiere quando un DragSource passa oltre l'elemento DragTarget

per richiamare la registrazione automatica degli elementi occorre richiamare DragManager.install nell'evento onload di window:

```
window.onload = function () {
	DragManager.install();
	DragManager.startCapture();
}
```

Il comportamento conseguente ai vari Drag and Drop verrà definito nelle funzioni di callback, ad esempio:

```
		function dragOver(srcElement) {
			this.style.background = '#EEE';
		}

		function dragOut(srcElement) {
			this.style.background = '';
		}

		function dragOn (srcElement){
			if(srcElement!=this) {
				if (this.dragGroup == srcElement.dragGroup) {
					this.swapNode(srcElement);
				}
			}
		}
```

in tutti e tre i casi l'argomento che riceve la funzione è l'elemento DragSource mentre il contesto in cui viene eseguita (il this) è l'elemento DragTarget. Ma la libreria supporta, come abbiamo detto, anche la pos-

sibilità di definire dinamicamente sorgente e target. La registrazione di un elemento come DragSource avviene semplicemente come:

```
var myElement = document.createElement('div');
DragManager.registerSource (myElement);
...
```

quella di un DragTarget sarà invece:

```
DragManager.registerTarget (myElement,dragCallback);
...
```

# 3.8 CHIAMATE A CODICE ESTERNO

## 85 Aggiungere dinamicamente script esterni alla pagina

Spesso capita di usare, in una pagina, molti script diversi che risiedono in vari files esterni, questo comporta lo scaricamento di tutto il materiale prima che la pagina sia disponibile. Utilizzando il DOM però è possibile anche aggiungere script esterni "on-demand" ovvero solo quando servono. Per provare la tecnica prepariamo un semplice script esterno chiamato sampleScript.js contenente una dichiarazione di variabile:

```
var message = "ciao";
```

Poi prepareremo il documento HTML inserendo questa funzione nel codice script:

```
    function loadScript(id, url,fxCallback) {
      // tag head
      var head = document.getElementsByTagName("head")[0];
       // se esiste già uno script con lo stesso ID lo rimuove
```

```
        // Crea un nuovo <script>
        Var oScript = document.createElement("script");
        // Imposta l'attributo src
        oScript.setAttribute("src", url);
        // Imposta l'attributo id dello script
        oScript.setAttribute("id",id);
        head.appendChild(oScript);
        // richiama la funzione
        fxCallback();
    }
```

Questa funzione crea dinamicamente un elemento <SCRIPT> e lo aggiunge a <HEAD> dopo aver impostato l'attributo src puntandolo al file esterno. Come terzo argomento abbiamo poi una funzione di callback che verrà eseguita dopo che lo script è stato creato. A questo punto, nel codice HTML creiamo un bottone associando l'onclick alla funzione test:

```
<button onclick="test()">Load Script</button>
```

La funzione test avrà il compito di caricare il nostro script esterno ed eseguire la funzione callback:

```
    function test (){
        loadScript('script1','sampleScript.js',callback)
    }
```

La funzione callback, infine controllerà (a intervalli di 100ms) che sia disponibile la variabile message definita nello script esterno e quindi mostrerà un messaggio con il contenuto:

```
    function callback (){
        //nel caso lo script non sia caricato ripete tra 100 ms
        if(typeof(message)=='undefined') window.setTimeout
```

```
                                        (callback,100);
        else alert(message);
    }
```

È importante che prima di richiamare variabili e oggetti importati dallo script esterno si controlli che questi siano effettivamente presenti perché il caricamento del file esterno è asincrono.

## 86 Eseguire una funzione in un'altra finestra

L'oggetto window, che rappresenta la pagina web è qualificato come oggetto di primo livello del DOM, tant'è vero che le funzioni e le variabili esterne ad altri oggetti sono in realtà metodi e oggetti di window. Scrivere:

```
var a=1;
equivale a:
```
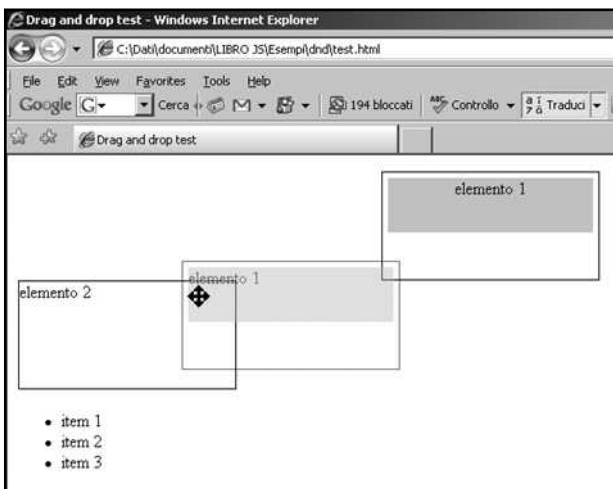
**Figura 84-1:** Il Drag and Drop in azione

```
window.a = 1;
o anche a:
window['a']=1;
```

Questa particolarità ci consente di usare un'utile tecnica, i riferimenti cross-window. Si definisca, ad esempio, un documento HTML con un frameset contenente un'unica frame:

```
<html>
<head>
  <script language="javascript" type="text/javascript">
        var user = "rossi";
   </script>
</head>
<frameset rows="*">
     <frame src="cross-window-frame1.html"/>
</frameset>
</html>
```

si noti che nello script viene definita la variabile user.
Si definisca poi il documento contenuto nella frame:

```
<html>
<head>
     <script language="javascript" type="text/javascript">
   window.onload = function () {
      document.getElementById("userName").innerHTML= 'L\'utente
                                                  corrente è ' + parent.user;
     }
   </script>
</head>
<body>
     Nome Utente definito nella finestra superiore:<br/>
```

```
    <div id="userName"></div>
</body>
</html>
```

Vedete come lo script, in onload, intercetti la variabile definita nella finestra superiore (parent) semplicemente come fosse (e in realtà lo è) una proprietà di quest'ultima. Allo stesso modo è possibile richiamare funzioni definite in altra finestra:

```
parent.sayHallo();
```

Nella gerarchia delle finestre parent rappresenta la finestra superiore, top la finestra di primo livello e self la finestra corrente. Naturalmente è possibile riferirsi anche a finestre al pari livello, se ad esempio il frameset ha due frame chiamate frame1 e frame2 , frame2 nello script si riferisce a frame1 come oggetto del suo parent, quindi per richiamare la variabile msg di frame1:

```
var msg = parent.frame1.msg;
```

Questa tecnica può essere utile sia per condividere metodi e variabili per tutte le frame inferiori (una specie di include implicito) che per condividere dei dati senza usare i Cookie.

# 3.9 I COOKIE

## 87 Recuperare informazioni dai Cookie

I Cookie sono semplici file di testo che risiedono nel computer dell'utente, al contrario di quello che avviene per tutti gli altri file Javascript può leggerli e scriverli (naturalmente solo quelli relativi al dominio corrente), essi possono essere inoltre temporanei (vengono distrutti a fine della sessione di navigazione) o persistenti (rimangono

oltre la sessione per un tempo definito). I Cookie vengono scritti nella forma nome=valore separati da punto e virgola, ad esempio:

```
user=rossi; id=2
```

Ecco una funzione che ci consente di leggere un Cookie dato il suo nome (oppure restituisce un valore di default):

```
function getCookie (name,defaultValue) {
    var nameEQ = name + "=";
    var ca = document.cookie.split(';');
    for(var i=0;i < ca.length;i++) {
        var c = ca[i];
        while (c.charAt(0)==' ') c = c.substring(1,c.length);
        if (c.indexOf(nameEQ) == 0) {
            var ckValue=c.substring(nameEQ.length,c.length);
            if(ckValue!='')    return ckValue;
            }
    }
    return defaultValue;
}
```

## 88 Impostare valori nei Cookie

Così com'è possibile leggere i Cookie, con Javascript si possono anche impostare:

```
function setCookie (name,value) {
    var expires = "";
    var days = arguments[2]; // 3° param expire
    var path = arguments[3] || "/"; // 4° param path
    if (days) {
        if(!isNaN(new Number(days))) {
            var date = new Date();
```

```
                    date.setTime(date.getTime()+(days*24*60*60*1000));
                    expires = "; expires="+date.toGMTString();
               }
          }
          var setValue = name+"="+value+expires+"; path=" + path;
          document.cookie = setValue;
     }
```

la funzione supporta anche 2 argomenti non dichiarati : la prima sono i giorni di persistenza del Cookie nel computer dell'utente, la seconda è il percorso dal quale il cookie è leggibile (potremmo rendere il Cookie visibile per tutto il sito, "/", o solo a partire da una determinata directory "/dir").

## 89 Rimuovere un valore dai Cookie

Per cancellare un valore dai Cookie basta impostare la scadenza ad una data già scaduta. Noi lo facciamo utilizzando la stessa funzione set Cookie con il valore di giorni di persistenza negativo:

```
function removeCookie (name){
          Cookies.set(name,"",-1);
}
```

## 90 Sapere se l'utente ha disabilitato i Cookie

Lo svantaggio dell'uso dei Cookie come fonte di dati è dato dal fatto che l'utente li può disabilitare, per conoscere se lo ha fatto (e magari avvertirlo con un avviso) abbiamo a disposizione la proprietà cookieEnabled dell'oggetto navigator:

```
if(!navigator.cookieEnabled) {
     alert('non hai abilitato i Cookie!');
}
```

# 3.10 LOCATION

## 91 Navigare verso un'altra pagina

L'oggetto location di window rappresenta l'url del documento caricato nel browser. Quello che non tutti sanno è che location è una proprietà di lettura e scrittura e che impostandola non un altro indirizzo il browser carica automaticamente il nuovo URL. In pratica con questa tecnica non siamo più vincolati ad usare solo l'elemento A per i link, qualsiasi elemento può essere un link:

```
<div onclick="window.location='nuovourl.html'">clicca qui</div>
proprio come:
<a href="nuovourl.html">clicca qui</div>
```

## 92 Conoscere i parametri della URL

Chi programma lato server è abituato a lavorare con i parametri passati da una pagina all'altra attraverso l'url, ad esempio: *http://www.google.it/search?q=parametri&sourceid=navclient-ff&ie=UTF-8* i parametri sono quella parte dell'URL dopo il "?":

```
q=parametri&sourceid=navclient-ff&ie=UTF-8
```

I parametri sono recuperabili attraverso location.search di window. Ecco come costruire un oggetto da questi parametri :

```
String.prototype.splitPairs=function(pairSeparator,innerSeparator){
   var pairs=this.split(pairSeparator);
   var result=new Object;
   for(var i=0; i<pairs.length;i++) {
      var kv=pairs[i].split(innerSeparator);
      if (kv.length!=0){
         var k= kv[0];
         var v= (kv.length>1)? kv[1] : null;
```

```
            result[k]=v;
        }
    }
    return  result;
}

 function getRequest () {
      var _request= {};
      if(location.search.length!=0) {
             q = location.search.substr(1);
             _request = q.splitPairs("&","=");
      }
      return _request;
}
```

Quindi, per recuperare un valore dai parametri URL basterà:

```
//URL: http://www.google.it/search?q=parametri&sourceid=
                                                    navclient-ff&ie=UTF-8
var r = getRequest();
alert(r.sourceid); // dà navclient-ff
```

# ACCESSO A FONTI DI DATI ESTERNE

La possibilità di accesso a fonti di dati esterne attraverso XMLHttpRequest o altre tecniche è quella che ci permette di svincolarci dal ciclo request/response di ricaricamento dell'intera pagina ed apre la strada ad un colloquio più "fluido" tra client e server evitando il ricaricamento continuo. L'insieme di queste tecnologie è comunemente noto come AJAX, sono un po' la moda del momento, ma la base su cui poggiano è talmente solida da aver portato Javascript da linguaggio per i "gadgets" decorativi del sito a vera e propria spina dorsale di applicazioni web.

## 93 Recupero di dati attraverso XMLHttpRequest

Il meccanismo di recupero di dati dal server attraverso XMLHttpRequest è simile un po' per tutti i browser più recenti, l'unica differenza è che mentre IE7, Firefox, Opera ed altri browser integrano in maniera nativa l'oggetto XMLHttpRequest, IE5 e IE6 devono ricorrere ad un oggetto activeX per cui, per instanziare l'oggetto si adotta questa tecnica "a cascata":

```
function getXMLHttpRequest (){
    if (typeof(XMLHttpRequest) != 'undefined') {
        return new XMLHttpRequest();
    }
else {
    return new ActiveXObject("Msxml2.XMLHTTP");
}
}
```

una volta creato l'oggetto comunque il funzionamento è analogo, se si desidera avere una risposta sincrona (cioè aspettare fino a quando il server non restituisce i dati) occorre usare:

```
    var req = getXMLHttpRequest();
```

```
	var method="POST"//o GET;
	req.open(method, "documento.xml", false);
	var body = "par1=abc&par2=1";
// body è il messaggio da inviare al server contenente i parametri
	if (method=="POST") {
		//settaggio degli header in caso di POST
		req.setRequestHeader ("Content-Type",
							"application/x-www-form-urlencoded");
}
	req.send(body);
	var result =  req.responseText; // il risultato dal server
```

più frequentemente però si usa la tecnica asincrona, che permette di non bloccare l'attività dell'utente mentre i dati si stanno caricando:

```
	var req = this.getXMLHttpRequest();
	var onreadystate = function (){
		if (req.readyState == 4) {
			try{
				if (req.status && req.status == 200)
{
				//i dati sono arrivati
				var data = req.responseText;
				...
				}
				else {
				//c'è un errore!
				var err = req.statusText;
				...
				}
			}
			catch(e){
			//tutti gli altri possibili errori
```

```
}
            }
            else {
                  //in caricamento...
            }
      }
      //assegnazione della funzione di callback prima definita
      req.onreadystatechange = onreadystate;
var method="POST"//o GET;
      req.open(method, "documento.xml", true);
      if (method=="POST") {
      req.setRequestHeader ("Content-Type",
                                          "application/x-www-form-urlencoded");
}
      var body = "par1=abc&par2=1";
      req.send(body);
```

Naturalmente risulterebbe piuttosto scomodo riscrivere tutto questo codice ogni volta che dobbiamo effettuare una richiesta al server, per cui risulta più conveniente incapsulare tutto in un oggetto:

```
var jsXML = new Object();

jsXML.getXMLHttpRequest = function(){
      if (typeof(XMLHttpRequest) != 'undefined') {
            return new XMLHttpRequest();
      }
else {
      return new ActiveXObject("Msxml2.XMLHTTP");
}
}
jsXML.setPost = function (req) {
      req.setRequestHeader ("Content-Type",
```

```
                                                "application/x-www-form-urlencoded");
}
jsXML.sendSync = function(url,body){
	var req = this.getXMLHttpRequest();
	method=(arguments[2])?arguments[2]:"POST";
	req.open(method, url, false);
	if (method=="POST") jsXML.setPost(req);
	req.send(body);
	return req.responseText;
}

jsXML.sendAsync = function (method,url,body,action,onfault,onwait) {
	var req = this.getXMLHttpRequest();
	var onreadystate = function (){
		if (req.readyState == 4) {
			try{
				if (req.status && req.status == 200)
{
					if(action!=null) action(req.responseText,req);
				}
				else {
					if(onfault!=null) onfault(req.statusText);
				}
			}
			catch(e){}
		}
		else {
			if(onwait!=null) onwait(req.readyState);
		}
	}
	req.onreadystatechange = onreadystate;
	req.open(method, url, true);
	if (method=="POST") jsXML.setPost(req);
```

```
    if(body==null) body='';
    req.send(body);
}
```

in questo modo per effettuare una richiesta sincrona al server basta richiamare:

```
var response=jsXML.sendSync("myServerPage.xml",
                              "par1=abc&par2=1","POST");
mentre, per la richiesta asincrona:
//callback per la risposta positiva
function responseSuccess(responseText){
    // fai qualcosa con responseText
}
//callback per l'errore
function responseError(errorMessage){
    // fai qualcosa con errorMessage
}
//callback per l'attesa
function responseWaiting(readyState){
    //sta caricando...
}
jsXML.sendAsync("POST","myServerPage.xml",
  "par1=abc&par2=1", responseSuccess, responseError, responseWaiting);
```

Come abbiamo visto i parametri vengono passati al server nella forma urlencoded e cioè a coppie nome/valore separate dal segno "&".
La risposta invece è una stringa di testo, naturalmente quasi mai sarà del testo statico, ma piuttosto del testo generato da una procedura lato server con dati magari provenienti da database.
Contrariamente a quanto si potrebbe pensare XMLHttpRequest non serve solo per recuperare dati in formato XML : il server può rispondere con qualunque stringa, sta a Javascript poi il compito di analizzarla ed agire

di conseguenza, tant'è vero che ultimamente è invalsa l'abitudine di far rispondere al server con stringhe contenenti codice Javascript in notazione contratta (c.d. JSON) che vengono valutate attraverso la funzione eval. Nella notazione JSON (JavaScript Object Notation) si utilizza una forma dichiarativa sintetica, ad esempio un oggetto di questo tipo:

```
var obj = new Object;
obj.prop1="s";
obj.prop2=new Array;
obj.prop2[0]=10;
obj.prop2[1]=20;
secondo le regole Javascript si può scrivere anche:
var obj = {'prop1':'s','prop2':[10,20]};
```

Il principio di JSON è recuperare i dati dal server in forma di espressione Javascript e poi utilizzarli valutandoli attraverso eval: eval(serverResponse). I vantaggi di JSON rispetto a XML sono:

- si trasmettono meno dati (una stringa JSON è generalmente più corta rispetto a un file XML, anche se le dimensioni di quest'ultimo si possono ridurre notevolmente impiegando gli attributi anziché gli elementi)
- non c'è bisogno di ricorrere all'interpretazione dell'XML in quanto il codice, dopo essere stato valutato, è come fosse stato scritto direttamente nel programma

gli svantaggi invece sono:

- per i dati da inviare dal client al server XML è un formato più robusto e affidabile
- se la struttura dei dati è molto profonda o se occorre effettuare delle selezioni su di essi XML dispone di XPath che in questo è molto efficiente.

- il flusso XML prodotto dal server potrà essere utilizzato anche da altri programmi mentre quello JSON no
- XML può essere trasformato in HTML attraverso XSLT

In sostanza potremmo dire che il tipo di risposta fornita dal server (XML, JSON o anche semplicemente puro testo) deve essere decisa secondo le circostanze e le esigenze del programma.

## 94 XMLHttpRequest e il cross-domain

Il principale problema di XMLHttpRequest sono le restrizioni di sicurezza al caricamento dei dati, se infatti la pagina contenente lo script che usa XMLHttpRequest si trova ad esempio all'indirizzo *http://www.miodominio.it/pagina.html* ed i dati da caricare si trovassero invece all'indirizzo *http://www.esterno.it/dati.xml* XMLHttpRequest genererà un errore (vedi **figura 94-1**). La ragione di tale restrizione consiste nel fatto che consentire l'accesso ad altri domini dal proprio è rischioso perché potrebbe consentire, da parte di malintenzionati, l'uso del dominio come base di attacco ad altri domini. Comunque a volte potrebbe essere indispensabile accedere a dati esterni al proprio dominio, come ad esempio nel caso di Web Service.
In questo caso la soluzione migliore consiste nel creare un cd. application proxy ovvero un'applicazione (ASP,PHP ecc...) che risiede nel nostro dominio e, lato server, ritrasmette la richiesta verso il dominio esterno, riceve la risposta e la ritrasmette a XMLHttpRequest (vedi **figura 94-2**). Un semplice script proxy in PHP 5 (con le estensioni CURL attivate) è:
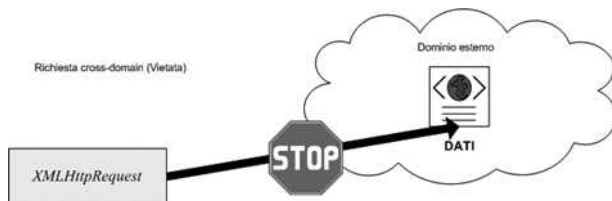
**Figura 94-1:** Errore nel recupero dati cross-domain
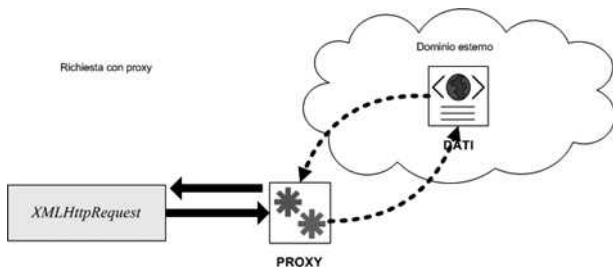
**Figura 94-2:** Recupero dati attraverso un application proxy

```
<?
// per motivi di sicurezza definisco
// SERVERHOST nel codice
define('SERVERHOST', 'http://www.esterno.it/data/');
// path : indirizzo relativo del documento da recuperare
$path = $_GET['path'];

// L'URL sarà SERVERHOST + path
$url = SERVERHOST.$path;
// Apri la sessione Curl
$session = curl_init($url);
// recupera i dati
curl_setopt($session, CURLOPT_HEADER, false);
curl_setopt($session, CURLOPT_RETURNTRANSFER, true);
// effettua la chiamata
$xml = curl_exec($session);

// scrive l'XML
header("Content-Type: text/xml");

echo $xml;
curl_close($session);
?>
```

Richiamabile, con le funzioni che abbiamo visto al punto 93:

```
jsXML.sendAsync('GET','proxy.php?file.xml','',successCallback,error
                                                            Callback);
```

Come alternativa a questa soluzione, se si usa Apache come Web Server, è possibile usare il modulo Proxy o il modulo Rewrite per re-indirizzare le richieste. Un'altra soluzione (che in realtà esclude l'uso di XMLHttpRequest) si basa sulla possibilità di caricamento dinamico di scripts attraverso il DOM (attraverso la tecnica descritta al punto 85), il principio è semplice: visto che uno script esterno può risiedere in un altro dominio e che lo script può essere generato dinamicamente dal server (con ASP,PHP ecc...) perché allora non recuperare i dati richiedendo al server un certo script e facendo in modo che questo risponda con i dati in formato JSON? La differenza rispetto a è che qui la procedura lato server può tranquillamente risiedere nel server esterno:

```
loadScript('script1', 'http://www.esterno.it/ottieniscript.php?req=1',
                                                            callback);
```

Questa tecnica sta cominciando ad essere abbastanza diffusa, tuttavia ritengo che i vantaggi (possibilità di accedere a domini esterni) non siano tali da compensare gli svantaggi (comunque occorre una procedura lato server per di più residente sull'altro dominio, il formato dei dati deve essere necessariamente JSON, la tecnica si adatta male a recuperare dati da Web Services ecc...).

## 95 Upload di dati

Mentre per quanto riguarda il recupero dei dati XMLHttpRequest rimane la soluzione primaria, per l'invio dei dati al server questa non sempre funziona. Non funziona, in particolare, quando al server dobbiamo inviare dei files in quanto l'unico modo di accedere ai file locali è l'uso dell'elemento <input type="file">. Dovremmo quindi rinunciare in questo ca-

so alla nostra sofisticata interfaccia Javascript e tornare al POST che ricarica l'intera pagina? Non necessariamente. Il trucco sta nel "postare" la form non verso un'altra pagina, ma verso una IFRAME nascosta all'interno della pagina stessa. In primo luogo prepareremo il codice HTML nella nostra pagina che contiene la FORM per inviare i file:

```
<html>
<body>
<div id="wait" style="display:none">Attendere prego</div>
<form action="upload.php" onsubmit="uploadData()" id="theForm"
target="processFrame" method="post" enctype="multipart/form-data">
    <div class="inputGroup">
    <div class="label">primo file</div>
    <input class="inputFile" type="file" name="file1">
    </div>
    <div class="inputGroup">
    <div class="label">secondo file</div>
    <input class="inputFile" type="file" name="file2">
    </div>
    <div class="submit"><input type="submit" value="invia"></div>
</form>
<iframe id="processFrame" frameborder="0" width="0" height="0"
src="about:blank" name="processFrame">
</iframe>
</body>
</html>
```

Fin qui nulla di particolarmente nuovo rispetto a quanto siamo abitati a fare normalmente: c'è una FORM che punta a un file (in questo caso PHP) che processerà i dati inviati con enctype impostato a "multipart/form-data" ecc... Notiamo però che nella pagina c'è anche una IFRAME con bordo, larghezza e altezza impostate a 0 (ovvero invisibile) e che l'attributo target (ovvero il "bersaglio") della FORM è proprio questa IFRAME. C'è infine

anche un DIV , con id "wait", con il messaggio "Attendere prego" nascosto attraverso l'attributo di stile "display:none". Prepariamo adesso, nella stessa directory, lo script PHP che processerà i dati (upload.php):

```
<html>
<head></head>
<body>
<script language="javascript" type="text/javascript">
<?
$uploaddir = '';
$count = 0;
$success = 0;
$errors = 0;
foreach($_FILES as $file) {
    if($file['size']>0) {
        $count++;
        if (move_uploaded_file($file['tmp_name'],
                                        $uploaddir . $file['name'])) {
            $success++;
        }
        else {
            $errors++;
        }
    }
}
print ("var count=$count ;\n");
print ("var success=$success ;\n");
print ("var errors=$errors ;\n");
?>
var parentWin = window.parent;
if (parentWin && parentWin['finishUpload']){
    parentWin.finishUpload(count,success,errors);
}
```

```
</script>
</body>
</html>
```

in pratica lo script effettua semplicemente il salvataggio dei file nella posizione impostabile attraverso la variabile $uploaddir e conta il numero dei file, quelli salvati con successo e quello dove è stato riscontrato un errore. Poi scrive queste variabili nel codice Javascript di ritorno. La pagina non ha interfaccia (ricordiamoci che viene eseguita in una IFRAME nascosta!) ma consiste in pratica solo in uno script Javascript; dopo che PHP ha scritto le tre variabili nello script, viene richiamata una funzione (finishUpload) che sarà definita nella pagina che ospita l'IFRAME. Quindi torniamo al file HTML contenente la FORM e aggiungiamo alla pagina il seguente script:

```
	<script language="javascript" type="text/javascript">
		function hide (id){
			var el = document.getElementById(id);
			el.style.display='none';
		}
		function show (id){
			var el = document.getElementById(id);
			el.style.display='';
		}

		function showWait (){
			show('wait');
			hide('theForm');
		}

		function showForm (){
			show('theForm');
			hide('wait');
```

```
        }
        //chiude il messaggio dopo 3 secondi
        function setTimeoutForWait (){
            setTimeout(showForm,3000);
        }
//chiamato da upload.php
        function finishUpload (count,success,errors){
            var wait = document.getElementById('wait');
            wait.innerHTML= "File trasmessi con successo:"
                                        + success + "<br/>" +
            "File con errori:" + errors;
            setTimeoutForWait();
        }
        //chiamato da onsubmit
        function  uploadData(){
            showWait();
        }

    </script>
```

Il meccanismo è molto semplice:

- quando l'utente invia i file, sull'evento onsubmit della FORM, viene chiamata la funzione uploadData
- uploadData mostra il messaggio di attesa e nasconde la FORM per impedire che l'utente possa effettuare altri invii
- il codice lato server viene eseguito nella IFRAME e produce una chiamata alla funzione finishUpload
- finishUpload aggiorna il pannello con il messaggio con un breve report sulla trasmissione dei file e, dopo 3 secondi, il pannello scompare e riappare la FORM per un nuovo invio

In questo modo la pagina principale non viene ricaricata e l'applicazione funziona proprio come farebbe con XMLHttpRequest.